

O'REILLY®

Wydanie V

Python

Wprowadzenie



Helion

Mark Lutz

O'REILLY®

Wydanie V

Python

Wprowadzenie



Helion

Mark Lutz

Mark Lutz

Python

Wprowadzenie

Wydanie V

*Przekład: Grzegorz Kowalczyk,
Andrzej Watrak, Anna Trojan,
Marek Pętlicki*

Tytuł oryginału: Learning Python, 5th Edition

Tłumaczenie: Grzegorz Kowalczyk (Przedmowa, rozdz. 1 – 29), Andrzej Watrak (rozdz. 30 – 41, dodatki) z wykorzystaniem fragmentów książki "Python. Wprowadzenie. Wydanie IV" w przekładzie Anny Trojan i Marka Pętlickiego

ISBN: 978-83-283-6151-5

© 2020 Helion SA

Authorized Polish translation of the English edition of Learning Python, 5th Edition ISBN 9781449355739 © 2013 Mark Lutz

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicielami.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA ul. Kościuszki 1c, 44-100 Gliwice tel. 32 231 22 19, 32 230 98 63 e-mail:
helion@helion.pl WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku! Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pytho5_ebook Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kod źródłowy przykładów wraz z rozwiązaniami ćwiczeń można pobrać ze strony:
<ftp://ftp.helion.pl/przyklady/pytho5.zip>

- [Poleć książkę](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » nasza społeczność](#)

Przedmowa

Jeżeli jesteś w księgarni i szukasz krótkiego opisu tej książki, polecam poniższy:

- *Python* to wieloparadygmatowy język programowania o ogromnych możliwościach, zoptymalizowany pod kątem produktywności programisty, czytelności kodu i jakości oprogramowania.
- *Niniejsza książka* jest kompleksowym i obszernym wprowadzeniem do języka Python. Jej celem jest pomoc w opanowaniu podstaw języka Python przed zastosowaniem ich w pracy. Podobnie jak wszystkie poprzednie edycje, ta książka ma służyć jako kompleksowe źródło wiedzy dla wszystkich nowych użytkowników Pythona, niezależnie od tego, czy będą używać Pythona 2.x, Pythona 3.x, czy obu tych wersji.
- To wydanie zostało zaktualizowane do wersji 3.3 oraz 2.7 Pythona i zostało znacznie rozszerzone, aby odzwierciedlić zmiany zachodzące obecnie w świecie Pythona.

W niniejszej przedmowie znajdziesz szczegółowy opis celów, zakresu i struktury tej książki. Jest to zupełnie opcjonalny podział, który jednak ma na celu przybliżenie Ci ogólnego zarysu tej książki, zanim zaczniesz czytać ją na dobre.

„Ekosystem” tej książki

Python to popularny język programowania z dziedziny open source, wykorzystywany w wielu różnych zastosowaniach zarówno w samodzielnnych programach, jak i skryptach. Jest darmowy, łatwo przenośny, ma duże możliwości i jest zarówno stosunkowo łatwy do nauczenia, jak i niezwykle przyjemny w użyciu. Programiści z każdego zakątku branży oprogramowania uznają, że koncentracja Pythona na wydajności programisty i jakości oprogramowania daje strategiczną przewagę w dużych i małych projektach.

Bez względu na to, czy jesteś doświadczonym programistą, czy też poczynającym użytkownikiem Pythona, celem niniejszej książki jest omówienie podstaw tego języka programowania. Po przeczytaniu tej książki powinieneś wiedzieć wystarczająco dużo o Pythonie, aby móc wykorzystywać go w wielu różnych dziedzinach.

Z założenia niniejsza książka jest praktycznym przewodnikiem skupiającym się ogólnie na języku *Python*, a nie jego specyficznych zastosowaniach. Tym samym przeznaczona jest do roli pierwszej części dwutomowego zbioru składającego się z następujących pozycji:

- *Learning Python* (w Polsce: *Python. Wprowadzenie*), czyli niniejszej książki, która koncentruje się na najważniejszych zagadnieniach związanych z Pythonem.
- *Programming Python* (i wiele innych) — pokazującej, co można zrobić z Pythonem po tym, jak się go nauczyłeś.

Taki podział jest celowy. Zakres i dziedziny zastosowania Pythona mogą się różnić w zależności od użytkownika, ale zapotrzebowanie na dobry opis podstaw języka programowania zawsze będzie takie samo. Oznacza to, że książki poświęcone dziedzinie aplikacji, takie jak *Programming Python*, kontynuują to, na czym kończy się niniejsza książka, wykorzystując szereg realistycznych, skalowalnych przykładów do omawiania roli Pythona w popularnych zastosowaniach, takich jak internet, graficzne interfejsy użytkownika (GUI), bazy danych czy

przetwarzanie tekstu. Dodatkowo książka *Python. Leksykon kieszonkowy* udostępnia materiały nieujęte tutaj i zaprojektowana jest w taki sposób, by uzupełniać niniejszą pozycję.

Ponieważ jednak niniejsza książka koncentruje się na fundamentach języka, jest w stanie przedstawić podstawy Pythona z większą głębią, niż wielu programistów widzi przy pierwszym podejściu do nauki języka programowania. Jej oddolne podejście i niezależne przykłady dydaktyczne mają na celu nauczenie czytelników całego języka krok po kroku.

Podstawowe umiejętności językowe, które zdobędziesz w tym procesie, będą miały zastosowanie do każdego napotkanego systemu oprogramowania opartego na Pythonie — czy to do dzisiejszych popularnych narzędzi, takich jak Django, NumPy i App Engine, czy innych, które mogą być zarówno częścią przyszłości Pythona, jak i Twojej kariery jako programisty.

Ponieważ cała książka opiera się na trzydniowym kursie praktycznym języka Python z quizami i ćwiczeniami, stanowi również znakomite, samodzielne wprowadzenie do języka. Chociaż w jej formacie brakuje nieco interakcji, jaką osiągnąć można w przekazywaniu wiedzy na żywo, rekompensuje to dodatkową głębią i elastycznością, którą może zapewnić tylko książka. Choć istnieje wiele sposobów korzystania z tej książki, ci, którzy przeczytali ją „od deski do deski”, zazwyczaj uznają ją za przybliżony odpowiednik semestralnej nauki języka Python.

O niniejszym, piątym wydaniu książki

Poprzednie, czwarte wydanie tej książki opublikowane w 2009 r. dotyczyło wersji 2.6 i 3.0 języka Python.^[1] Omawialiśmy w nim wiele, czasem niekompatybilnych zmian wprowadzonych ogólnie w linii 3.x Pythona. Wprowadziłliśmy także nowy samouczek skoncentrowany na programowaniu zorientowanym obiektowo oraz nowe rozdziały poruszające zaawansowane tematy, takie jak przetwarzanie tekstu Unicode, dekoratory i metaklasy, oparte zarówno na materiałach szkoleniowych z kursów, które prowadzę, jak i ewolucji „najlepszych praktyk” programowania w języku Python.

Piąta edycja tej książki została ukończona w 2013 roku. Jest rozbudowaną aktualizacją wcześniejszej wersji, która obejmuje zarówno Pythona 3.3, jak i 2.7, czyli najnowsze wersje Pythona w liniach 3.x i 2.x. Książka zawiera wszystkie zmiany językowe wprowadzone w obu liniach od czasu opublikowania poprzedniej edycji i została dopracowana pod kątem omawianych aktualizacji i ulepszenia przykładów. A konkretnie wygląda to tak:

- Zagadnienia związane z linią 2.x Pythona zostały zaktualizowane i zawierają opisy funkcji, takich jak słowniki i wyrażenia składane, które wcześniej były dostępne wyłącznie w wersji 3.x, ale z czasem zostały zaimplementowane również w wersji 2.7.
- Zagadnienia związane z linią 3.x Pythona zostały rozbudowane o nową składnię instrukcji `yield` i `raise`, model `__pycache__` kodu bajtowego; pakiety przestrzeni nazw od wersji 3.3; pakiet PyDoc z trybem dla wszystkich przeglądarek; zmiany w literałach i sposobie przechowywania ciągów Unicode oraz nowy program uruchamiający Windows dostarczany z wersją 3.3.
- Dodano różne zagadnienia związane z nowymi lub rozszerzonymi mechanizmami, takimi jak obsługa formatu JSON, moduł `timeit`, pakiet PyPy, metoda `os.popen`, generatory, rekurencja, słabe referencje, atrybuty i metody takie jak `__mro__`, `__iter__`, `super`, `__slots__`, metaklasy, deskryptory, funkcja `random`, pakiet Sphinx i wiele innych. Wprowadzonych zostało również wiele modyfikacji związanych z ogólnym polepszeniem kompatybilności przykładów i opisów z wersją 2.x.

W tym wydaniu dodaliśmy również nowe wnioski w postaci rozdziału 41. (o ewolucji Pythona), dwa nowe dodatki (opisujące ostatnie zmiany w Pythonie i nowy program uruchamiający dla systemu Windows) oraz jeden nowy rozdział (o testowaniu wydajności i efektywności kodu, zawierający rozszerzoną wersję przykładu mierzącego czas działania kodu). W dodatku C

znajdziesz krótkie podsumowanie zmian, jakie zaszły w Pythonie między poprzednim a obecnym wydaniem książki, a także odnośniki do odpowiednich zagadnień w książce. W tym dodatku podsumowano również początkowe różnice między wersjami 2.x i 3.x, które zostały po raz pierwszy uwzględnione w poprzedniej edycji, chociaż niektóre z nich, takie jak na przykład nowego typu klasy, były dostępne już wcześniej, a po prostu stały się obowiązkowe w wersji 3.x (więcej na temat tego, co oznacza ów tajemniczy „x”, powiemy za chwilę).

Zgodnie z ostatnim punktem na poprzedniej liście ta edycja została również rozszerzona o pełniejsze pokrycie bardziej *zaawansowanych funkcji językowych* — które w ostatniej dekadzie wielu z nas bardzo starało się zignorować jako opcjonalne, ale które teraz są powszechnie spotykane w kodzie Pythona. Jak zobaczymy, narzędzia te zwiększą możliwości Pythona, ale także podnoszą poprzeczkę dla nowych użytkowników i mogą zmieniać zasięg i definicję Pythona. Ponieważ zawsze możesz się spotkać z takimi zmianami, w tej książce omawiamy je bezpośrednio, zamiast udawać, że nie istnieją.

Pomimo aktualizacji zachowano w tym wydaniu większość struktury i zawartości poprzedniej edycji, a cała książka jest nadal zaprojektowana jako kompleksowe źródło wiedzy dla Pythona w wersjach 2.x i 3.x. Chociaż koncentrujemy się przede wszystkim na użytkownikach Pythona 3.3 i 2.7, czyli odpowiednio najnowszej wersji z linii 3.x i prawdopodobnie ostatniej wersji z linii 2.x, historyczna perspektywa tej książki sprawia, że jest ona również odpowiednia dla użytkowników starszych wersji Pythona, które są nadal w szerokim użyciu.

Chociaż nie można przewidzieć przyszłości, w tej książce staraliśmy się wyróżnić pewne fundamentalne zagadnienia, które obowiązują od prawie dwóch dekad i prawdopodobnie będą miały zastosowanie również w *przyszłych* wersjach Pythona. Jak zwykle na stronie internetowej oryginalnej wersji tej książki znajdziesz informacje o aktualizacjach w języku Python, które mogą wpływać na opisywane tutaj zagadnienia. Więcej szczegółowych informacji na ten temat znajdziesz również w dokumentach *What's new* (co nowego?) dołączanych do dokumentacji kolejnych wersji, ponieważ po ukazaniu się tej książki Python z pewnością zmieni się jeszcze nie raz i nie dwa.

Python 2.x i 3.x kiedyś

Ponieważ ma to duży wpływ na treść tej książki, musimy najpierw powiedzieć kilka słów o historii Pythona 2.x/3.x. Kiedy w 2009 roku powstało *czwarte wydanie* tej książki, Python zaczynał być dostępny w dwóch wersjach:

- Wersja 3.0 była pierwszą wersją z nowej linii oznaczonej jako 3.x, która była unowocześnioną, ale niekompatybilną z linią 2.x mutacją języka Python.
- Wersja 2.6 zachowała kompatybilność wsteczną z ogromną ilością istniejącego kodu Pythona i była najnowszą wersją z linii znanej jako 2.x.

Chociaż wersja 3.x była w dużej mierze tym samym językiem, nie pozwalała na uruchamianie praktycznie żadnego kodu napisanego dla poprzednich wydań. Było to spowodowane tym, że w wersji 3.x:

- Narzucono model Unicode, co miało szerokie konsekwencje dla ciągów znaków, plików i bibliotek.
- Znaczaco zwiększo rolę iteratorów i generatorów, jako część pełniejszego paradygmatu funkcyjnego.
- Wprowadzono nowy styl klas, które łączą się z typami, ale stają się bardziej wydajne i złożone.
- Zmieniono wiele podstawowych narzędzi i bibliotek, a część całkowicie usunięto lub zastąpiono innymi.

Sama mutacja polecenia `print` z instrukcji do postaci funkcji, choć może być estetyczna i uzasadniona, spowodowała, że prawie wszystkie programy napisane w języku Python dla

poprzednich wersji przestały działać poprawnie w nowej wersji. Pomijając wiele innych, strategicznych aspektów nowej wersji, narzucenie w wersji 3.x obowiązkowego modelu Unicode i nowych klas oraz wszechobecnych generatorów zostało przeprowadzone w celu ułatwienia tworzenia rozbudowanych systemów i złożonych aplikacji.

Chociaż wielu użytkowników niemal od razu zaczęło postrzegać Pythona 3.x zarówno jako ulepszenie, jak i przyszłość Pythona, wersja 2.x była nadal bardzo szeroko stosowana i przez długie lata miała być obsługiwana równolegle z Pythonem 3.x. Większość używanego w tamtym okresie kodu Pythona była przygotowana dla wersji 2.x, a migracja do wersji 3.x wydawała się kształtać jako powolny proces.

Python 2.x i 3.x obecnie

Obecna, piąta edycja tej książki została napisana w 2013 roku; w tym czasie Python został zaktualizowany do wersji 3.3 i 2.7, ale historia związana ze współistnieniem linii 2.x/3.x nadal pozostała w dużej mierze *niezmieniona*. W rzeczywistości Python jest obecnie światem istniejącym w dwóch wymiarach (liniach), z wieloma użytkownikami korzystającymi zarówno z wersji 2.x, jak i 3.x, zgodnie z ich celami programowymi, zależnościami i potrzebami. Dla wielu nowych użytkowników wybór między wersjami 2.x i 3.x sprowadza się do wyboru między korzystaniem z istniejących, działających i sprawdzonych programów a nowocześniejszą wersją języka programowania. Chociaż wiele dużych pakietów Pythona zostało już przeniesionych do wersji 3.x, wiele innych wciąż jest dostępnych tylko w wersji 2.x.

Przez niektórych obserwatorów Python 3.x jest teraz postrzegany jako *poligon* do odkrywania nowych pomysłów, podczas gdy wersja 2.x jest postrzegana jako *wypróbowany i prawdziwy* Python, który co prawda nie ma wszystkich funkcji wersji 3.x, ale nadal jest bardziej wszechobecny. Inni użytkownicy nadal widzą Pythona 3.x jako perspektywę, która wydaje się być forsowana przez głównych deweloperów [2]. Z drugiej strony inicjatywy takie jak PyPy — do dziś jedyna implementacja Pythona tylko dla wersji 2.x, która oferuje wręcz oszałamiające ulepszenie wydajności — stanowią o przyszłości linii 2.x, jeżeli nie są wręcz osobną frakcją.

Jeżeli jednak odsuniemy wszelkie sentymenty na bok, prawie pięć lat po pojawienniu się na rynku wersja 3.x nadal musi jeszcze walczyć o zastąpienie wersji 2.x, która obecnie jest nadal częściej pobierana z witryny python.org niż wersja 3.x, choć sytuacja odwrotna wydawałaby się zupełnie naturalna. Takie statystyki są oczywiście podatne na zmiany, ale po pięciu latach wskazują jednak na pewną akceptację wersji 3.x. Istniejąca baza oprogramowania 2.x w wielu przypadkach przebiją rozszerzenia językowe 3.x. Co więcej, bycie ostatnim w linii 2.x sprawia, że wersja 2.7 jest *de facto standardem*, odpornym na ciągłe tempo zmian w linii 3.x — pozytywnym dla tych, którzy szukają stabilnej bazy, a negatywnym dla tych, którzy szukają wzrostu i ciągłego rozwoju.

Osobiście uważam, że dzisiejszy świat Pythona jest wystarczająco duży, aby pomieścić zarówno wersje 3.x, jak i 2.x; wydają się one spełniać różne cele i przemawiać do różnych grup użytkowników, co jest już często spotykane w innych rodzinach języków programowania (na przykład C i C++ od dawna współistnieją, chociaż różnią się od siebie bardziej niż Python 2.x i 3.x). Co więcej, ponieważ obie linie są mimo wszystko do siebie bardzo podobne, umiejętności zdobyte podczas uczenia się jednej z linii Pythona znaczco ułatwiają migrację do tej drugiej linii, zwłaszcza jeżeli korzystasz z zasobów omawiających obie wersje i różnice między nimi (tak jak ta książka). W rzeczywistości, o ile rozumiesz specyfikę i różnice pomiędzy poszczególnymi liniami Pythona, często będziesz w stanie napisać kod poprawnie działający w obu wersjach.

Z drugiej strony istniejący podział stanowi poważny *dylemat* zarówno dla programistów, jak i autorów książek. Chociaż w książce łatwiej byłoby udawać, że Python 2.x nigdy nie istniał, i skoncentrować się tylko na wersji 3.x, nie zaspokoi to potrzeb dużej grupy użytkowników Pythona. W Pythonie 2.x napisano do tej pory ogromną ilość kodu, który jest nadal używany i szybko nie zniknie. Podczas gdy początkujący użytkownicy tego języka mogą i powinni skupić się na wersji 3.x Pythona, każdy, kto nadal używa kodu napisanego w przeszłości, musi dziś

nadal utrzymywać kontakt ze światem Pythona 2.x. Ponieważ może upływać jeszcze wiele lat, zanim wszystkie biblioteki i rozszerzenia firm trzecich zostaną przeniesione do Pythona 3.x, takie rozdrojenie może nie być całkowicie tymczasowe.

Omawiamy zarówno wersję 2.x, jak i 3.x

Aby odpowiedzieć na tę dychomię i sprostać potrzebom wszystkich potencjalnych użytkowników, niniejsze wydanie książki zostało uaktualnione w taki sposób, by omawiać zarówno Pythona 3.3, jak i Pythona 2.7 (a także późniejsze wydania z linii 3.x oraz 2.x). Niniejsza książka przeznaczona jest dla programistów wykorzystujących Pythona 2.x, programistów używających Pythona 3.x oraz wszystkich innych programistów, którzy utknęli gdzieś pomiędzy tymi dwiema liniami.

Oznacza to, że możesz wykorzystać tę książkę do poznania obu linii Pythona. Choć skupiamy się tutaj przede wszystkim na wersji 3.x, różnice w stosunku do Pythona 2.x i narzędzia z tej wersji są po drodze opisywane z przeznaczeniem dla programistów korzystających ze starszego kodu. Choć obie wersje są w dużej mierze podobne, różnią się na kilka istotnych sposobów, co będziemy podkreślać w miarę omawiania poszczególnych komponentów języka.

Przykładowo w większości kodów będziemy wykorzystywać wywołanie `print` z Pythona 3.x, jednak opiszemy także instrukcję `print` z wersji 2.x, tak abyś łatwo mógł zrozumieć starszy kod i korzystać z przenośnych technik wyświetlania danych, które działają w obu liniach Pythona. Będziemy także swobodnie wprowadzać nowe opcje tego języka programowania, takie jak instrukcja `nonlocal` z wersji 3.x czy metoda `format` ciągów znaków, dostępna od wersji odpowiednio 2.6 i 3.0, a także wskazywać, kiedy takie rozszerzenia nie są obecne w jeszcze wcześniejszych wersjach.

Poprzez analogię obecne wydanie książki dotyczy również innych wersji Pythona z linii 2.x i 3.x, chociaż niektóre przykłady kodu 2.x mogą nie działać w starszych wersjach. Na przykład dekoratory klas są dostępne od wersji 2.6 i 3.0 Pythona i nie można ich używać w starszych wersjach Pythona 2.x, które jeszcze nie miały zaimplementowanego tego mechanizmu. Więcej szczegółowych informacji i podsumowanie ostatnich zmian w wersjach 2.x i 3.x znajdziesz w dodatku C.

Której wersji Pythona powinieneś użyć?

Wybór wersji może być uzależniony od decyzji Twojej firmy czy organizacji, ale jeżeli dopiero zaczynasz przygodę z Pythonem i sam się uczysz, możesz samodzielnie zastanowić się, którą wersję powinieneś zainstalować. Odpowiedź zależy od Twoich celów. Oto kilka sugestii dotyczących wyboru.

Kiedy wybrać wersję 3.x: nowe funkcje, ewolucja

Jeżeli uczysz się Pythona od podstaw i nie musisz korzystać ze starszego kodu, powinieneś rozpocząć naukę od razu od Pythona 3.x. W tej wersji usunięto sporo problemów, które nagromadziły się przez lata w tym języku, zachowując jednak przy tym wszystkie oryginalne idee leżące u jego podstaw i dodając sporo ciekawych, nowych narzędzi. Na przykład uniwersalny model Unicode oraz szerokie zastosowanie generatorów i technik funkcyjnych są postrzegane przez wielu użytkowników wersji 3.x jako jej zdecydowane atuty. Wiele popularnych bibliotek i narzędzi Pythona jest już dostępnych lub niebawem będzie dostępnych dla wersji 3.x, zwłaszcza biorąc pod uwagę ciągłe udoskonalenia i aktualizacje tej linii Pythona. Nowe rozszerzenia i aktualizacje języka pojawiają się obecnie już tylko w wersji 3.x, co sprawia, że to właśnie ta wersja jest najbardziej odpowiednia, ale powoduje też, że kanon tego języka podlega ustawniczym zmianom — jest to jednak nieodłącznie związane z procesem ustawnicznego rozwoju i wdrażania nowych, lepszych mechanizmów i rozwiązań.

Kiedy wybrać wersję 2.x: istniejący kod, stabilność

Jeżeli będziesz używać systemu opartego na języku Python 2.x, linia 3.x może nie być dla Ciebie najlepszym wyborem. Przekonasz się jednak, że ta książka również dotyczy Twoich problemów i pomoże Ci, jeżeli w niedalekiej przyszłości przeprowadzisz migrację do wersji 3.x. Przekonasz się również, że dołączysz w ten sposób do dużej społeczności użytkowników, choć nadal regularnie spotyka się przydatne oprogramowanie napisane w Pythonie 2.x. Co więcej, w przeciwieństwie do wersji 3.x wersja 2.x nie jest już rozbudowywana i aktualizowana — co jest wadą lub zaletą, w zależności od tego, kogo zapytasz. Nie ma nic złego w używaniu i pisaniu kodu dla wersji 2.x, ale powinieneś już pomału migrować swoje projekty do wersji 3.x i korzystać z jej ciągłej ewolucji. Przyszłość Pythona jest ciągle otwarta i zależy w dużej mierze od jego użytkowników, w tym Ciebie.

Kiedy wybrać obie wersje: kod neutralny dla wersji

Prawdopodobnie najlepszą nowiną jest to, że podstawy Pythona są takie same w obu wersjach — 2.x i 3.x różnią się w sposób, który wielu użytkowników uzna za nieznaczny, a ta książka została tak zaprojektowana, aby pomóc Ci nauczyć się obu wersji. W rzeczywistości, o ile rozumiesz dzielące je różnice, często łatwo jest napisać kod neutralny dla wersji, który działa na obu liniach Pythona — w książce znajdziesz wiele przykładów takiego kodu. Wskazówki dotyczące migracji z wersji 2.x do wersji 3.x i wskazówki dotyczące pisania kodu dla obu wersji Pythona znajdziesz w dodatku C.

Bez względu na to, którą wersję lub wersje wybierzesz, Twoje umiejętności zostaną przeniesione bezpośrednio tam, gdzie zaprowadzi Cię praca z językiem Python.

	<p><i>Kilka słów o znaku „x”:</i> W tej książce oznaczenia „3.x” i „2.x” używane są w odniesieniu do wszystkich wersji w tych dwóch liniach języka Python. Na przykład 3.x obejmuje wersje od 3.0 do 3.3 oraz wszystkie przyszłe wersje 3.x; określenie 2.x oznacza wszystko od 2.0 do 2.7 (i przypuszczalnie tyle, bo wszystko wskazuje na to, że linia 2.x zostanie zakończona na wersji 2.7). Bardziej szczegółowe numery wersji są wymieniane, gdy omawiane zagadnienie dotyczy tylko danego tematu (np. literałów zbiorów w wersji 2.7 czy programu uruchamiającego i przestrzeni nazw w wersji 3.3). Notacja ta może czasami być zbyt szeroka — niektóre funkcje oznaczone tutaj jako 2.x mogą nie być dostępne we wcześniejszych wersjach z linii 2.x, które są dziś rzadko używane, ale nadal należą do linii 2.x, która jest już z nimi od 19 lat. Określenie linii 3.x jest nieco bardziej precyzyjne i obejmuje okres ostatnich 11 lat (wersja 3.0 Pythona została wydana w grudniu 2008 roku).</p>
---	--

Wymagania wstępne dla użytkowników tej książki

Podanie niezbędnych wymagań do pracy z tą książką jest praktycznie niemożliwe, ponieważ jej użyteczność i wartość mogą zależeć zarówno od motywacji Czytelnika, jak i od jego doświadczenia. Zarówno prawdziwi początkujący, jak i zaprawieni w bojach weterani programowania z powodzeniem korzystali z niej w przeszłości. Jeżeli masz motywację do nauki języka Python i chcesz poświęcić czas oraz skupić się na pracy, ta książka prawdopodobnie będzie dla Ciebie odpowiednia.

Ile czasu zajmuje nauka języka Python? Chociaż różni się to w zależności od użytkownika, ta książka sprawdza się najlepiej, gdy się ją po prostu czyta. Niektórzy Czytelnicy mogą jej używać jako źródła informacji na żądanie, ale większość osób, które chcą nauczyć się Pythona, powinna przygotować się na spędzenie z nią co najmniej kilku tygodni, o ile nie miesiące, w zależności

od tego, jak ściśle będą podążać za omawianymi w niej przykładami. Jak wspominaliśmy już wcześniej, jest to mniej więcej odpowiednik semestralnego kursu języka Python.

Oczywiście jest to tylko szacunkowy czas potrzebny do nauki samego Pythona i opanowania umiejętności programowania niezbędnych do jego prawidłowego używania. Chociaż ta książka może wystarczyć do realizacji podstawowych zadań skryptowych, czytelnicy mający nadzieję na rozwój swoich umiejętności programowania i kontynuowania kariery programisty powinni po jej przeczytaniu spodziewać się konieczności poświęcenia dużej ilości czasu na zbieranie doświadczenia w tworzeniu dużych projektów i ewentualnie na pracę z innymi książkami, takimi jak *Programming Python*^[3].

To może nie być najlepsza wiadomość dla osób, które chcieliby szybko nabrać biegłości w posługiwaniu się Pythonem, ale programowanie wcale nie jest trywialną umiejętnością (pomimo tego, co mogłeś gdzieś usłyszeć!). Dzisiejszy Python i ogólnie cały proces tworzenia oprogramowania stanowią zarówno ogromne wyzwanie, jak i dają wielką satysfakcję, z nawiązką wynagradzającą długie miesiące nauki, studiowania obszernych książek, tomów dokumentacji i mozołnego zdobywania doświadczenia praktycznego. Oto kilka wskazówek dotyczących korzystania z tej książki dla czytelników po obu stronach tego spektrum doświadczeń:

Dla doświadczonych programistów

Masz początkową przewagę i możesz szybko poruszać się po rozdziałach wstępnych; nie powinieneś jednak całkowicie pomijać podstawowych zagadnień, a być może nawet od czasu do czasu będziesz musiał nieco zwolnić, aby doczytać to czy owo. Ogólnie rzecz biorąc, doświadczenie w tworzeniu innych programów lub skryptów przed rozpoczęciem pracy z tą książką może być bardzo pomocne ze względu na analogie, których może ona dostarczyć. Z drugiej strony odkryłem również, że wcześniejsze doświadczenia w programowaniu mogą nieco utrudniać naukę Pythona ze względu na pewne zakorzenione nawyki z innych języków (bardzo łatwo można poznać, kto jest programistą Javy lub C++, analizując kod pierwszych, samodzielnego napisanych przez nich klas Pythona!). Właściwe korzystanie z możliwości Pythona wymaga pełnego przyjęcia jego filozofii. Niniejsza książka koncentruje się na kluczowych koncepcjach tego języka i została tak zaprojektowana, aby pomóc Ci nauczyć się „używać Pythona” w Pythonie.

Dla początkujących użytkowników

Dzięki tej książce również możesz się nauczyć języka Python, a także samego programowania, ale być może będziesz musiał trochę ciężej pracować i uzupełniać wiedomości danymi z innych źródeł. Jeżeli nie uważasz się za programistę, to mimo to ta książka również będzie dla Ciebie bardzo przydatna, z tym że powinieneś spokojnie i powoli przyswajać kolejne zagadnienia i po drodze uważnie wykonywać wszystkie opisywane przykłady i ćwiczenia. Pamiętaj też, że w tej książce poświęcamy więcej czasu na naukę samego Pythona niż na podstawy programowania. Jeżeli zgubisz się w natłoku omawianych zagadnień, przed rozpoczęciem pracy z tą książką powinieneś zapoznać się ze wstępem do programowania. Na witrynie internetowej Pythona znajdziesz łącza prowadzące do wielu bardzo ciekawych i przydatnych zasobów dla początkujących.

Formalnie rzecz biorąc, niniejsza książka została pomyślana jako *pierwszy podręcznik Pythona dla każdego początkującego użytkownika* tego języka programowania. Być może nie jest idealną książką dla osób, które nigdy nie miały kontaktu z komputerem (na przykład dlatego, że nie będziemy tracić czasu na objaśnianie sposobu działania komputera), ale nie zakładamy również z góry, że czytelnik zna jakieś języki programowania czy ma jakiekolwiek wykształcenie w dziedzinie informatyki.

Z drugiej strony, nie zamierzam obrażać czytelników, nadając książce łatkę „dla opornych” — cokolwiek by to znaczyło. Korzystając z Pythona, można bardzo łatwo robić wiele przydatnych rzeczy, a ta książka pokaże Ci, jak to zrobić. W niektórych miejscach będziemy porównywać

możliwości Pythona i innych języków programowania, takich jak C, C++ czy Java, ale jeżeli nigdy nie korzystałeś z tych języków, możesz takie uwagi spokojnie zignorować.

Struktura tej książki

W tej sekcji znajdziesz krótki przegląd zawartości i głównych celów poszczególnych części tej książki. Jeżeli chcesz od razu przejść do zagadnień związanych z Pythonem, możesz spokojnie pominąć tę sekcję (lub zamiast niej przejrzeć spis treści). Mamy jednak nadzieję, że w tak obszernej książce zamieszczenie takiego „streszczenia” jest całkiem dobrym pomysłem.

Z założenia każda część książki obejmuje jeden z głównych obszarów funkcjonalnych języka i składa się z rozdziałów koncentrujących się na konkretnym temacie lub aspekcie zagadnień danej części. Ponadto każdy rozdział kończy się *quizami* i odpowiedziami, a każda część kończy się większymi ćwiczeniami, których rozwiązania przedstawiono w dodatku D.

	<p><i>Praktyka ma znaczenie:</i> zdecydowanie zalecamy, abyś zapoznawał się z quizami i ćwiczeniami zawartymi w tej książce, i jeżeli to możliwe, wykonywał wszystkie omawiane przykłady. W programowaniu nic nie zastąpi samodzielnego sprawdzania i testowania tego, czego się dowiedziałeś. Niezależnie od tego, czy robisz to z tą книгą, czy z własnego projektu, kluczowym czynnikiem jest tworzenie prawdziwego kodu i wykorzystywanie przedstawianych tutaj pomysłów i koncepcji.</p>
---	---

Jak już wspominaliśmy wcześniej, książka ta prezentuje Pythona od podstawowych zagadnień po te bardziej złożone. W miarę postępów omawiane zagadnienia stają się coraz bardziej złożone. Na przykład klasy Pythona są w większości pakietami funkcji przetwarzających typy wbudowane. Po poznaniu wbudowanych typów i funkcji klasy stają się tematem stosunkowo łatwym do opanowania. Ponieważ każda część opiera się na poprzedzających zagadnieniach, większość czytelników uznaje, że *czytanie tej książki po kolejnych rozdziałach*, jest najbardziej sensownym rozwiązaniem. Poniżej zamieszczamy zestawienie oraz krótki opis głównych części naszej książki:

Część I

Rozpoczynamy od ogólnych informacji na temat języka Python oraz odpowiedzi na najczęściej zadawane na początku pytania — dlaczego warto używać tego języka programowania, do czego jest on przydatny i tym podobne. W pierwszym rozdziale omówione zostaną najważniejsze informacje dotyczące technologii, co powinno dać każdemu pewien kontekst sytuacyjny. Później rozpoczyna się część techniczna książki, w której zajmiemy się tym, w jaki sposób my sami wykonujemy programy oraz jak robi to Python. Celem tej części książki jest udostępnienie wystarczającej ilości informacji wstępnych, które pozwolą na swobodne śledzenie późniejszych przykładów oraz ćwiczeń.

Część II

Następnie rozpoczynamy naszą wycieczkę po języku Python, skupiając się na początku na najważniejszych wbudowanych typach obiektów i tym, co można z nimi zrobić: liczbach, listach, słownikach i tak dalej. Dzięki takim obiektom możesz wiele zrobić i znajdziesz je w sercu każdego skryptu napisanego w języku Python. To najważniejsza część książki, która stanowi podstawę dla kolejnych rozdziałów. W tej części przyjrzymy się również typom dynamicznym oraz ich interfejsom — kolejnym kluczowym zagadnieniom w Pythonie.

Część III

W kolejnej części omówione zostaną *instrukcje*, czyli kod, którego używamy do tworzenia i przetwarzania obiektów w Pythonie. Zaprezentujemy również ogólny model składni

Pythona. Choć w tej części skupimy się głównie na składni, omówimy również kilka powiązanych narzędzi, takich jak system PyDoc, przedstawimy podstawowe koncepcje związane z iteracjami, a także alternatywne możliwości tworzenia kodu.

Część IV

Ta część jest początkiem omawiania struktur programistycznych wyższego poziomu. *Funkcje* okazują się być prostym sposobem na spakowanie kodu do ponownego użycia i unikanie nadmiarowości kodu. W tej części omówimy reguły związane z zasięgami, techniki przekazywania argumentów, wspomnijmy o funkcjach lambda i poruszymy wiele innych zagadnień. Ponownie przyjrzymy się iteratorom z perspektywy programowania funkcyjnego, wprowadzimy generatory definiowane przez użytkownika i pokażemy, jak mierzyć czas działania kodu Pythona w celu zmierzenia i oszacowania jego wydajności.

Część V

Moduły służą w Pythonie do organizowania instrukcji i funkcji w większe komponenty; w tej części pokazane zostanie, jak moduły się tworzy, przeładowuje i jak się ich używa. Przyjrzymy się również niektórym bardziej zaawansowanym zagadnieniom, w tym pakietom modułów, ich przeładowywaniu, importowaniu względem, pakietom przestrzeni nazw dostępnym od wersji 3.3 oraz zmiennej `__name__`.

Część VI

W tej części omówimy element z dziedziny programowania zorientowanego obiektowo — klasy. *Klasa* to opcjonalny i bardzo wydajny sposób strukturyzowania kodu w celu dopasowania go do własnych potrzeb i późniejszego ponownego wykorzystania, co niemal w naturalny sposób wpływa na zminimalizowanie nadmiarowości kodu. Jak się zresztą przekonasz, klasy wykorzystują koncepcje omówione we wcześniejszych rozdziałach książki, a programowanie zorientowane obiektowo w Pythonie polega w dużej mierze na wyszukiwaniu zmiennych w połączonych obiektach. Jak zresztą zostanie to pokazane, programowanie zorientowane obiektowo jest w Pythonie opcjonalne, choć wielu użytkowników uważa, że jest dużo prostsze niż w innych językach programowania i może znaczco zmniejszyć czas potrzebny na tworzenie programów, w szczególności w przypadku długofalowych projektów.

Część VII

Omawianie podstawowych zagadnień języka Python kończymy przedstawieniem modelu obsługi wyjątków w Pythonie, wykorzystywanych w tym modelu instrukcji oraz krótkim przeglądem narzędzi programistycznych, takich jak narzędzia do testowania i debugowania, które staną się bardziej przydatne, kiedy zaczniesz pisać większe programy. Choć wyjątki są narzędziem stosunkowo prostym w użyciu, ta część pojawia się po omówieniu klas, ponieważ wszystkie wyjątki definiowane przez użytkownika powinny teraz być w Pythonie klasami. Omówimy tutaj także bardziej zaawansowane tematy, takie jak menedżery kontekstu.

Część VIII

W przedostatniej części książki omówimy niektóre bardziej zaawansowane zagadnienia. Zapoznamy się z łańcuchami Unicode oraz ciągami bajtowymi, narzędziami do zarządzania atrybutami, takimi jak właściwości i deskryptory, dekoratorami funkcji oraz klas, a także metaklasami. Rozdziały te są lekturą opcjonalną, ponieważ nie wszyscy programiści muszą znać zagadnienia, o których one traktują. Z drugiej strony osoby, które muszą przetwarzać teksty napisane w wielu językach, dane binarne lub odpowiedzialne są za tworzenie interfejsów API wykorzystywanych przez innych programistów, powinny w tej części znaleźć dla siebie coś ciekawego. Przykłady zamieszczone w tej części są również większe niż w pozostałych częściach i mogą służyć jako materiał do samodzielnej nauki.

Dodatki

Książka kończy się zestawem czterech dodatków, które zawierają wskazówki dotyczące instalowania i używania Pythona na różnych platformach, omawiają nowy program uruchamiający dla systemu Windows, który jest dostarczany z Pythonem 3.3, podsumowują zmiany w Pythonie, których dotyczyły ostatnie wydania, i przedstawiają rozwiązania ćwiczeń zamieszczanych na końcach poszczególnych części. Rozwiązania quizów końcowych z poszczególnych rozdziałów znajdują się na końcach tych rozdziałów.

Więcej szczegółowych informacji na temat zawartości książki znajdziesz w spisie treści.

Czym nie jest ta książka

Na przestrzeni lat z pewnością zgromadziła się całkiem dużą grupą czytelników, którzy z nadzieję oczekiwali, że ta książka będzie omawiała zagadnienia, które znajdowały się daleko poza jej zakresem, zatem teraz, gdy już napisaliśmy, czym jest ta książka, chcemy również jasno wyjaśnić, czym ona nie jest:

- Niniejsza książka to podręcznik do samodzielnej nauki Pythona, a nie leksykon tego języka.
- Niniejsza książka dotyczy samego języka Python, a nie jego zastosowań, standardowych bibliotek czy narzędzi firm trzecich.
- Niniejsza książka jest kompleksowym omówieniem poszczególnych zagadnień, a nie przeglądem tematów luźno związanych z Pythonem.

Ponieważ wymienione kwestie są kluczowe dla treści tej książki, chcielibyśmy powiedzieć jeszcze kilka słów na ich temat.

Ta książka nie jest leksykonem ani przewodnikiem po konkretnych zastosowaniach Pythona

Niniejsza książka to *podręcznik do nauki języka Python*, a nie leksykon czy przewodnik po jego zastosowaniach. Jest to zabieg celowy: *współczesny Python* — z wbudowanymi typami, generatorami, domknięciami, komponentami składanymi, obsługą Unicode, dekoratorami oraz mieszaną proceduralnych, obiektowych i funkcyjnych paradymatów programowania — sprawia, że kanon tego języka jest niezmiernie istotnym zagadnieniem sam w sobie, a także warunkiem wstępny wszelkiej przyszłej pracy w Pythonie, niezależnie od projektów i zastosowań, w których go używasz. Poniżej znajdziesz kilka sugestii innych źródeł i zasobów dotyczących Pythona:

Zasoby referencyjne

Jak wspominaliśmy już wcześniej, do wyszukania interesujących Cię tematów możesz użyć indeksu i spisu treści, ale niniejsza książka nie jest leksykonem tego języka. Jeżeli szukasz zasobów referencyjnych do Pythona (a większości czytelników wcześniej czy później będą one potrzebne), proponujemy zajrzeć do wspomnianej wcześniej pozycji, *Python. Leksykon kieszonkowy*, będącej znakomitym uzupełnieniem niniejszej książki, jak również innych podręczników i dokumentacji Pythona, które znajdziesz na stronie <http://www.python.org> i wielu innych. Dokumentacja na witrynie Pythona jest zawsze bezpłatna, zawsze aktualna i dostępna zarówno w internecie, jak i na komputerze po zainstalowaniu Pythona.

Aplikacje i biblioteki

Jak również już wspominaliśmy wcześniej, ta książka nie jest przewodnikiem po konkretnych zastosowaniach Pythona, takich jak internet, interfejsy graficzne czy programowanie systemowe. Przez analogię obejmuje to biblioteki i narzędzia używane w

aplikacjach; chociaż korzystamy tutaj z niektórych standardowych bibliotek i narzędzi, takich jak `timeit`, `shelve`, `pickle`, `struct`, `json`, `pdb`, `os`, `urllib`, `re`, `xml`, `random`, *PyDoc* czy *IDLE*, to nie są one oficjalnie objęte głównym zakresem tej książki. Jeżeli szukasz bardziej szczegółowych informacji na te tematy i masz już doświadczenie w pracy z Pythonem, jako uzupełnienie polecamy między innymi książkę *Programming Python*. Pamiętaj jednak, że założeniem tej książki jest, że dobrze znasz język podstawowy. W każdej dziedzinie inżynierii, włączając w to tworzenie oprogramowania, znajomość tematu i doświadczenie są niezbędnymi czynnikami.

To nie jest krótka historia dla ludzi, którzy się spieszą

Jak widać na podstawie jej rozmiarów, ta książka również nie oszczędza na szczegółach: przedstawia pełny kanon języka Python, a nie tylko krótkie spojrzenie na jego uproszczony podzbiór. Po drodze opisuje również zasady programowania, które są niezbędne do tworzenia dobrego kodu w języku Python. Jak wspominaliśmy, jest to książka na długie tygodnie, czy nawet miesiące pracy, mająca na celu przekazanie umiejętności na takim poziomie, jaki można uzyskać z pełnego kursu poświęconego nauce tego języka.

Jest to również zabieg celowy. Oczywiście wielu czytelników tej książki nie musi zdobywać umiejętności programistycznych na pełną skalę, a niektórzy potrafią nauczyć się korzystać z Pythona w sposób fragmentarny. Jednak ze względu na fakt, że w prawdziwym kodzie programów możesz napotkać dowolne konstrukcje języka Python, żadna część tej książki nie powinna być dla Ciebie opcjonalna. Co więcej, nawet zwykli użytkownicy i hobbyści piszący proste skrypty muszą znać podstawowe zasady tworzenia oprogramowania, aby dobrze kodować czy po prostu poprawnie korzystać ze wstępnie zakodowanych narzędzi.

Ta książka ma na celu zaspokojenie obu tych potrzeb — *języka i zasad programowania* — na tyle dogłębnie, aby były przydatne. W końcu jednak przekonasz się, że nawet bardziej zaawansowane narzędzia Pythona, takie jak programowanie funkcyjne i zorientowane obiektywnie, są stosunkowo łatwe do opanowania, gdy nauczysz się podstaw języka — a tak będzie, jeżeli będziesz pracował z tą книгą rozdział po rozdziale.

Ta książka jest tak liniowa, jak na to pozwala Python

A skoro mowa o kolejności czytania, to w tym wydaniu staraliśmy się również zminimalizować odwołania do przodu, choć zmiany w Pythonie 3.x uniemożliwiają to w niektórych przypadkach (w rzeczywistości wersja 3.x wydaje się zakładać, że znasz już Pythona podczas jego nauki!). Aby nie być gołosłownym:

- Wyświetlanie, sortowanie, metoda `format` ciągów znaków i niektóre wywołania `dict` opierają się na argumentach ze słowami kluczowymi.
- Listy kluczy słowników, testy oraz wywołania funkcji `list` używane w wielu narzędziach implikują zastosowanie mechanizmu *iteracji*.
- Użycie instrukcji `exec` do uruchamiania kodu zakłada teraz, że znasz obiekty plików i interfejsy.
- Kodowanie nowych wyjątków wymaga zastosowania klas i znajomości podstaw programowania zorientowanego obiektywnie.
- I tak dalej — nawet podstawowe *dziedziczenie* wymaga kontaktu z zaawansowanymi zagadnieniami, takimi jak *metaklasy* i *deskryptory*.

Pythona nadal najlepiej uczyć się, przechodząc od prostych zagadnień do coraz bardziej zaawansowanych, a w takim scenariuszu *liniowy postęp* jest nadal najbardziej sensowny. Mimo to niektóre zagadnienia mogą wymagać pewnych nieliniowych dygresji i przeskakiwania do innych tematów. Aby je zminimalizować, w tej książce będziemy wskazywać takie zależności w miarę ich występowania i starać się maksymalnie złagodzić ich wpływ.



A jeżeli masz mało czasu: chociaż do opanowania języka Python niezbędna jest wytrwałość i cierpliwość, niektórzy czytelnicy mogą dysponować po prostu ograniczonym czasem. Jeżeli chcesz rozpocząć od krótkiej wycieczki po Pythonie, proponujemy rozdział 1., rozdział 4., rozdział 10. i rozdział 28. (i być może 26.) — krótki przegląd najważniejszych zagadnień, który — mniejmy nadzieję — rozbudzi Twoje zainteresowanie bardziej kompletną opowieścią w dalszych częściach książki. Ogólnie rzecz biorąc, ta książka jest celowo ułożona w ten sposób, aby ułatwić przyswajanie przedstawianego w niej materiału — po wprowadzaniu zawsze następuje bardziej szczegółowe omówienie, dzięki czemu możesz zacząć od pobicznego przeglądu i z czasem pogłębiać swoją wiedzę. Nie musisz czytać całej książki od razu, ale jej metodyczne podejście ma na celu pomóc w szybkim przyswojeniu prezentowanych zagadnień.

O przykładach zamieszczonych w książce

Ogólnie rzecz biorąc, w niniejszej książce zawsze staraliśmy się zachować pewną neutralność zarówno w odniesieniu do wersji Pythona, jak i platform — jest zaprojektowana tak, aby była użyteczna dla wszystkich użytkowników Pythona. Niemniej jednak, ponieważ z biegiem czasu w Pythonie zachodzą znaczące zmiany, a poszczególne platformy mogą bardzo różnić się od siebie, musimy opisać konkretne systemy, które zobacysz w działaniu w większości prezentowanych tutaj przykładów.

Wersje Pythona

To już piąte wydanie tej książki i wszystkie zawarte w niej przykłady programów oparte są na wersjach 3.3 i 2.7 języka Python. Ponadto wiele prezentowanych przykładów działa we wcześniejszych wersjach linii 3.x i 2.x, a uwagi dotyczące historii zmian we wcześniejszych wersjach można znaleźć w całej książce.

Ponieważ nasza książka koncentruje się na podstawach Pythona, można zakładać, że większość prezentowanych tu zagadnień nie ulegnie znaczącym zmianom w przyszłych wersjach tego języka. Większość materiału tej książki ma zastosowanie również do starszych wersji Pythona, choć nie zawsze — to oczywiste, że kiedy spróbujesz użyć rozszerzeń dodanych po wydaniu używanej przez Ciebie starszej wersji, nie będą one działały. Zasadniczo możemy więc przyjąć, że najnowszy Python jest zawsze najlepszym Pythonem.

Ponieważ książka skupia się na podstawach tego języka, większość omawianych zagadnień ma zastosowanie również do dystrybucji *Jython* oraz *IronPython*, czyli implementacji Pythona odpowiednio w Javie i platformie .NET, a także innych implementacji Pythona, takich jak *Stackless* i *PyPy* (opisanych w rozdziale 2.). Takie alternatywne dystrybucje różnią się głównie szczegółami użytkowania, a nie kanonem języka.

Platformy

Przykłady w tej książce były uruchamiane na ultrabookach działających pod kontrolą systemów *Windows 7* i *8*, chociaż przenośność Pythona sprawia, że jest to mało istotne, szczególnie w tej książce skoncentrowanej na podstawach języka. W książce znajdziesz kilka zrzutów ekranu z systemu Windows, w tym okna wiersza polecenia, wskaźniki instalacji czy opis nowego programu uruchamiającego Pythona dla Windows, ale jest to związane głównie z tym, że większość początkujących użytkowników Pythona prawdopodobnie zacznie swoją przygodę właściwie na platformie Windows.

Znajdziesz tutaj również kilka szczegółów uruchamiania Pythona na innych platformach, np. Linuksie, takich jak np. zastosowanie wierszy *shebang #!*. Ale jak przekonasz się w rozdziale 3. I dodatku B, program uruchamiający w wersji 3.3 Pythona sprawia, że jest to nawet jeszcze bardziej przenośna technika.

Pobieranie kodów źródłowych dla tej książki

Kod źródłowy przykładów wraz z rozwiązaniami ćwiczeń można pobrać z serwera FTP wydawnictwa Helion (<ftp://ftp.helion.pl/przykłady/pytho5.zip>); odnośnik do materiałów znajduje się również na stronie polskiego wydania książki (<http://helion.pl/ksiazki/pytho5.htm>).

Znajdziesz tam kody przykładów z tej książki oraz instrukcje użytkowania pakietu, więc pominiemy teraz większość szczegółów. Oczywiście omawiane przykłady są dostosowane do zawartości poszczególnych rozdziałów, a do korzystania z nich potrzebna jest ogólna wiedza na temat uruchamiania programów w języku Python. Więcej szczegółowych informacji na temat uruchamiania programów znajdziesz w rozdziale 3., więc teraz musisz uzbroić się w pewną cierpliwość (lub po prostu zająrzyć do tego rozdziału).

Konwencje wykorzystywane w książce

W książce wykorzystywane są następujące konwencje typograficzne:

Kursywa

Oznacza adresy URL, adresy poczty elektronicznej, nazwy plików, ścieżki plików oraz służy do wyróżniania nowych pojęć, kiedy są one wprowadzane po raz pierwszy.

Czcionka o stałej szerokości

Wykorzystywana do oznaczania zawartości plików, a także danych wyjściowych poleceń, modułów, metod, instrukcji i samych poleceń.

Pogrubiona czcionka o stałej szerokości

Wykorzystywana we fragmentach kodu w celu wyróżnienia poleceń lub tekstu wpisywanych przez użytkownika. Czasami służy również do wyróżniania części kodu.

Pochylona czcionka o stałej szerokości

Oznacza tekst, który powinien być zastąpiony przez podane przez użytkownika wartości, a także tekst komentarzy w kodzie.

<Czcionka o stałej szerokości>

Oznacza element, który powinien być zastąpiony prawdziwym kodem.

	Taka ikona oznacza wskazówkę, sugestię lub ogólną uwagę odnoszącą się do tekstu.
	Taka ikona ta oznacza ostrzeżenie lub uwagę.

Od czasu do czasu w wybranych rozdziałach znajdziesz także ramki i przypisy, których zawartość jest często opcjonalna, ale zapewnia dodatkowy kontekst dla prezentowanych tematów. Ramki, takie jak „Warto pamiętać: wycinki” w rozdziale 7., często zawierają przykłady zastosowania omawianych zagadnień i komponentów.

Podziękowania

Pracując w roku 2013 nad piątym wydaniem tej książki, nie mogłem się powstrzymać od pewnej retrospekcji. Od dwudziestu jeden lat używam i promuję Pythona, od osiemnastu lat piszę o nim książkę, a od szesnastu lat prowadzę szkolenia z tego języka. Pomimo upływu czasu nadal jestem pod wielkim wrażeniem tego, jak wielki sukces osiągnął Python w tym okresie. Język ten rozwinął się w sposób, o którym większość nas we wczesnych latach dziewięćdziesiątych nawet nie śniła. I choć może to brzmieć jak egoistyczny wywód zapatrzonego w siebie autora, mam nadzieję, że wybaczysz mi kilka słów podsumowania, gratulacji i podziękowań.

Trochę wspomnień

Historia mojego związku z Pythonem poprzedza zarówno Pythona 1.0, jak i sieć WWW (i sięga czasów, kiedy instalacja oznaczała pobieranie kodu źródłowego, konsolidowanie, kompilowanie i trzymanie kciuków, żeby wszystko to jakoś zadziałało). Kiedy w 1992 roku po raz pierwszy odkryłem Pythona jako sfrustrowany programista C++, nie miałem pojęcia, jaki będzie to miało wpływ na następne dwie dekady mojego życia. Dwa lata po napisaniu pierwszego wydania książki *Programming Python* (O'Reilly 1995) zacząłem podróżować po całym kraju i świecie, prowadząc szkolenia z Pythona dla początkujących użytkowników i ekspertów. Od pierwszego wydania książki *Python. Wprowadzenie* (*Learning Python*, O'Reilly 1999) stałem się niezależnym trenerem i autorem zajmującym się tylko Pythonem — co było w dużej mierze zaśługą błyskawicznie rosnącej popularności tego języka.

Do tej pory mam już na koncie trzynaście książek na temat Pythona (wliczając pięć wydań tej książki i po cztery wydania dwóch innych), które według moich obliczeń sprzedają się już w ponad 400 tysiącach egzemplarzy. Uczę Pythona od ponad półtorej dekady i dotychczas odbyłem około dwustu sześćdziesięciu sesji szkoleniowych dla studentów ze Stanów Zjednoczonych, Europy, Kanady oraz Meksyku, których w sumie było już ponad cztery tysiące. Poza tym, że zbierałem mile w programach dla klientów linii lotniczych, wszystkie szkolenia pomagały mi ciągle ulepszać zawartość tej książki, a także pozostałych pozycji. Prowadzone szkolenia pozwalały mi na dopracowywanie książek i odwrotnie, w wyniku czego moje książki są ściśle powiązane z tym, co dzieje się na moich zajęciach, i mogą służyć jako realna alternatywa dla nich.

Co się tyczy samego Pythona, w ostatnich latach urósł on do rangi jednego z pięciu czy dziesięciu najczęściej używanych języków programowania na świecie (w zależności od źródła, na które się powołujemy, i kiedy się na nie powołujemy). Ponieważ będziemy omawiać status Pythona w pierwszym rozdziale tej książki, odłożę resztę tej historii do tego czasu.

Podziękowania dla Pythona

Ponieważ nauczanie uczy również samych nauczycieli, jak uczyć, ta książka wiele zawdzięcza szkoleniom, które prowadziłem. Z tego powodu chciałbym podziękować wszystkim *studentom*, którzy wzięli udział w moich kursach w ciągu ostatnich szesnastu lat. Oprócz zmian w samym Pythonie to Wasze reakcje i informacje zwrotne bardzo pomogły w ukształtowaniu tego tekstu (nie ma nic bardziej konstruktywnego od zobaczenia na żywo, jak cztery tysiące początkujących użytkowników powtarza te same błędy!). Ostatnie wydania tej książki zawdzięczają wiele zmian

przede wszystkim ostatnim turnusom kursów, choć każda grupa, która miała ze mną zajęcia od 1997 roku, w pewien sposób pomogła udoskonalić tę książkę. Chciałbym podziękować klientom, którzy zaprosili mnie do prowadzenia zajęć w Dublinie, Mexico City, Barcelonie, Londynie, Edmonton i Puerto Rico; takie doświadczenia były jednym z najwspanialszych przeżyć w mojej karierze.

Ponieważ pisanie uczy pisarzy pisać, książka ta wiele zawdzięcza również użytkownikom. Chcę podziękować wszystkim *Czytelnikom*, którzy przez ostatnie 18 lat poświęcali czas, aby przedstawić swoje sugestie, zarówno przez internet, jak i osobiście. Wasze opinie były również istotne dla tej książki i stały się istotnym czynnikiem sukcesu, korzyścią, która wydaje się być nieodłączną częścią świata otwartego oprogramowania. Komentarze rozciągają się od „Powinieneś mieć zakaz pisania książek” do „Niech Bóg ci błogosławi za napisanie tej książki”; jeżeli w takich sprawach możliwe jest osiągnięcie konsensusu, to prawdopodobnie leży on gdzieś pomiędzy tymi dwoma opiniami, choć, parafrazując cytat z powieści Tolkiena: ta książka jest wciąż zbyt krótka.

Chciałbym również wyrazić swoją wdzięczność wszystkim, którzy przyczynili się do powstania tej książki. Wszystkim tym, którzy przez lata pomagali w tworzeniu tej książki jako solidnego produktu — w tym jej redaktorom, składaczom, marketingowcom, korektorom technicznym i merytorycznym i nie tylko oraz wydawnictwu O'Reilly — za to, że dało mi szansę na pracę nad trzynastoma projektami książek; to była naprawdę świetna zabawa (nawet jeżeli czasami czułem się jak w filmie *Dzień świstaka*).

Dodatkowe podziękowania należą się całej społeczności Pythona; podobnie jak większość systemów open source, Python jest rezultatem wielu heroicznych wysiłków. Ogromnym wyróżnieniem było dla mnie obserwowanie, jak Python rozwija się od nowego języka skryptowego do szeroko wykorzystywanego narzędzia, używanego w jakimś stopniu przez niemal wszystkie firmy i organizacje tworzące oprogramowanie. Pomijając różnice zdań na tematy techniczne, bycie częścią tego przedsięwzięcia było ekscytującym przeżyciem.

Chciałbym również podziękować mojemu pierwszemu redaktorowi z wydawnictwa O'Reilly, nie żyjącym już Frankowi Willisonowi. Ta książka była w dużej mierze jego pomysłem. Miał głęboki wpływ zarówno na moją karierę, jak i na sukces Pythona, gdy był to nowy, mało znany język programowania; to jest spuścizna Franka, którą pamiętam i będę pamiętała za każdym razem, gdy kusi mnie, by nadużywać słowa „tylko”.

Podziękowania osobiste

Na koniec kilka osobistych podziękowań. Dla zmarłego Carla Sagana za zainspirowanie osiemnastoletniego dziecka z Wisconsin. Dla mojej mamy za jej odwagę. Dla mojego rodzeństwa za prawdy, które można znaleźć w muzealnych wydaniach komiksu „Fistaszki”. Dla książki *The Shallows* za bardzo mi potrzebne przebudzenie.

Dla mojego syna Michaela i córek Samanthy i Roxanne za to, kim jesteście. Nie jestem pewien, kiedy zdążyliście dorosnąć, ale jestem dumny z tego, jak to zrobiliście, i nie mogę się doczekać, kiedy zobaczymy, dokąd zaprowadzi Was życie.

I dla mojej żony Very — za cierpliwość, pomoc w korektach, dietetyczną colę i precle. Cieszę się, że w końcu Cię znalazłem. Nie wiem, co przyniesie mi następne pięćdziesiąt lat, ale wiem, że chcę je spędzić z Tobą.

— Mark Lutz *Amongst the Larch* wiosna 2013

[1] Krótkotrwała trzecia edycja z 2007 r. obejmowała Pythona 2.5 oraz jego uproszczony — i krótszy — jednowierszowy świat Pythona. Więcej informacji na temat historii tej książki można znaleźć na stronie <http://learning-python.com/books>. Z biegiem lat ta książka urosła do rozmiarów i złożoności wprost proporcjonalnych do rozwoju własnego Pythona. W dodatku C znajdziesz informację o tym, że w samej wersji 3.0 Pythona zaimplementowanych zostało 27

nowych mechanizmów i 57 zmian w języku, które zostały przynajmniej pokrótkę opisane w książce, a w Pythonie 3.3 ten trend jest kontynuowany. Dzisiejszy programista korzystający z Pythona ma do czynienia z dwiema niekompatybilnymi liniami, trzema głównymi paradygmatami, mnóstwem zaawansowanych narzędzi i ogromną nadmiarowością funkcji i mechanizmów, z których większość nie dzieli się równo między liniami 2.x i 3.x. To jednak wcale nie jest tak zniechęcające, jak może się na pierwszy rzut oka wydawać (wiele narzędzi to wariacje na temat), a wszystkie dostępne funkcje i mechanizmy są dobrze przemyślanymi komponentami w całościowym, kompleksowym świecie Pythona.

[2] Zgodnie z oficjalnym komunikatem z dniem 1 stycznia 2020 roku Python 2.x przestał być wspierany — *przyp. tłum.*

[3] Standardowe zastrzeżenie: napisałem tę i inne wspomniane wcześniej książki z założeniem, że będą stanowić razem zestaw: *Python. Wprowadzenie* to omówienie podstaw języka, *Python Programming* to wprowadzenie do tworzenia aplikacji oraz *Python. Leksykon kieszonkowy* jako towarzyszka dwóch pozostałych. Wszystkie trzy książki wywodzą się w pewnym sensie z wydanego w 1995 roku dzieła *Programming Python*. Zachęcam również gorąco do zapoznania się z wieloma innymi, dostępnymi dziś książkami na temat Pythona (przestałem sprawdzać dostępne książki, kiedy w witrynie Amazon.com naliczyłem ich ponad 200, a końca nie było widać; przy czym nie obejmowało to książek na tematy pokrewne, takie jak Django). Mój własny wydawca wprowadził niedawno na rynek książki poświęcone instrumentacji, przetwarzaniu danych, App Engine, analizie numerycznej, przetwarzaniu języka naturalnego, MongoDB, AWS i bardziej specyficznych domenom, które być może będziesz chciał zbadać po opanowaniu podstaw języka Pythona. Dzisiejszy zakres zastosowań Pythona jest zbyt obszerny, aby można go było zamknąć w kilku książkach.

Część I Wprowadzenie

Rozdział 1. Pytania i odpowiedzi dotyczące Pythona

Osoby, które kupiły niniejszą książkę, wiedzą zapewne, czym jest Python i dlaczego warto się go nauczyć. Jeśli tak nie jest, być może nie dowiedzą się tego, dopóki nie opanują tego języka, czytając resztę książki i wykonując parę pierwszych projektów. Zanim jednak przejdziemy do szczegółów, pierwszy rozdział książki poświęcimy krótkiemu omówieniu elementów składających się na popularność Pythona. Aby ułatwić Ci zrozumienie tego, czym jest Python, rozdział ten przygotowaliśmy w formie sesji odpowiedzi na pytania, które są najczęściej zadawane przez początkujących adeptów tego języka.

Dlaczego ludzie używają Pythona?

Ponieważ w tej chwili istnieje tyle języków programowania, jest to chyba pierwsze pytanie, które zadają osoby początkujące. Biorąc pod uwagę, że obecnie na świecie jest około miliona użytkowników Pythona, trudno jest na nie odpowiedzieć w precyzyjny sposób. Wybór narzędzi programistycznych jest często uzależniony od osobistych preferencji czy unikalnych ograniczeń danego projektu.

Jednak po przeprowadzeniu w ostatnich dwunastu latach około dwustu sześćdziesięciu kursów z Pythona, których uczestnikami było ponad cztery tysiące osób, okazało się, że pewne powody takiego stanu rzeczy często się powtarzają. Najważniejszymi czynnikami podawanymi przez użytkowników Pythona były najczęściej:

Jakość oprogramowania

Dla wielu osób nacisk położony na czytelność, spójność i jakość oprogramowania odróżnia Pythona od innych języków skryptowych. Kod w Pythonie został zaprojektowany w taki sposób, by był *czytelny* i tym samym — by można go było z łatwością utrzymywać i używać ponownie w o wiele większym stopniu, niż dzieje się to w przypadku innych języków skryptowych. Spójność kodu w Pythonie sprawia, że łatwo jest go zrozumieć, nawet jeśli samemu nie jest się jego autorem. Co więcej, Python obsługuje bardziej zaawansowane mechanizmy pozwalające na *ponowne wykorzystanie kodu*, takie jak programowanie zorientowane obiektowo i programowanie funkcyjne.

Wydajność programistów

Python wielokrotnie zwiększa wydajność i produktywność programistów w porównaniu z językami kompilowanymi czy o statycznych typach, takimi jak C, C++ czy Java. Kod w Pythonie stanowi średnio *jedną trzecią do jednej piątej* rozmiaru jego odpowiednika w C++ czy Javie. Oznacza to mniejszą liczbę znaków do wpisania, a także mniejszą ilość kodu do sprawdzenia i utrzymania w przyszłości. Programy napisane w Pythonie działają natychmiast, bez konieczności długiej komplikacji i korzystania z narzędzi zewnętrznych, co jeszcze bardziej zwiększa szybkość tworzenia kodu.

Przenośność programów

Większość programów napisanych w Pythonie działa bez zmian na *wszystkich najważniejszych platformach*. Przeniesienie Pythona z Linuksa na system Windows ogranicza się na ogół do skopiowania kodu skryptu między komputerami. Co więcej, Python oferuje wiele opcji kodowania przenośnych graficznych interfejsów użytkownika, programów dostępu do bazy danych czy systemów webowych. Nawet interfejsy systemów operacyjnych, wraz z uruchamianiem programów i przetwarzaniem katalogów, są w Pythonie tak przenośne, jak tylko można sobie życzyć.

Obsługa bibliotek

Python instalowany jest wraz z ogromnym zbiorem wbudowanych i przenośnych opcji, zwany *biblioteką standardową*. Biblioteka ta obsługuje mnóstwo zadań programistycznych na poziomie aplikacji, od dopasowywania wzorców po skrypty sieciowe. Dodatkowo istnieją rozszerzenia do Pythona w postaci zarówno niewielkich bibliotek, jak i ogromnej liczby programów obsługujących aplikacje. Istnieją narzędzia służące do konstruowania witryn internetowych, programowania numerycznego, dostępu do portu szeregowego, tworzenia gier i wielu innych funkcji. Przykładowo rozszerzenie NumPy jest opisywane jako darmowy i bardziej rozbudowany odpowiednik systemu programowania numerycznego Matlab.

Integracja komponentów

Skrypty Pythona z łatwością komunikują się z innymi częściami aplikacji, wykorzystując do tego liczne mechanizmy integracyjne. Taka integracja pozwala na wykorzystywanie Pythona jako narzędzia do rozszerzania i dostosowywania produktów do własnych potrzeb. Obecnie Python potrafi wywoływać biblioteki języków C i C++, a kod w Pythonie można wywoływać z programów w tych językach. Python integruje się z komponentami języków Java i .NET, potrafi komunikować się za pomocą platform takich jak COM czy Silverlight, a także (za pośrednictwem sieci) z interfejsami takimi, jak SOAP, XML-RPC i CORBA oraz potrafi sterować urządzeniami przez port szeregowy. Nie jest narzędziem działającym w oderwaniu od innych.

Przyjemność

Ze względu na łatwość w użyciu oraz wbudowany zbiór narzędzi Python sprawia, że programowanie to *bardziej przyjemność niż niemiły obowiązek*. Choć ta korzyść może być trudna do zdefiniowania, jej wpływ na wydajność programistów trudno przecenić.

Z tych czynników pierwsze dwa (jakość oraz wydajność) są najprawdopodobniej najważniejszymi korzyściami dla większości użytkowników Pythona i zasługują na bardziej szczegółowe przedstawienie.

Jakość oprogramowania

W Pythonie celowo zaimplementowano prostą i czytelną składnię, a także bardzo spójny model programowania. Jak podkreśla slogan jednej z ostatnich konferencji na temat Pythona, rezultat takiego działania jest taki, że Python zdaje się „pasować do naszego sposobu myślenia” — co oznacza, że możliwości tego języka pozostają ze sobą spójne i są naturalną konsekwencją niewielkiej liczby podstawowych koncepcji. To sprawia, że język ten łatwo jest zrozumieć, łatwo też nauczyć się go i zapamiętać. W praktyce programiści Pythona, kiedy czytają lub tworzą kod, nie muszą się ciągle odwoływać do podręczników i dokumentacji. Python to spójnie zaprojektowany system, który — w opinii wielu osób — pozwala na tworzenie zaskakująco spójnego i jednolitego kodu.

Filozofia Pythona zakłada minimalizm. Oznacza to, że choć istnieje wiele sposobów wykonania jednego zadania, zazwyczaj istnieje jedna oczywista droga, kilka mniej oczywistych alternatyw i niewielki zbiór spójnych interakcji w każdym obszarze tego języka. Co więcej, Python nie podejmuje za nas (jednych słusznych) decyzji — kiedy jakaś interakcja jest niejasna,

preferowane są jawne interwencje. W filozofii Pythona jawne jest lepsze od niejawnego, a proste — od skomplikowanego^[1].

Poza takimi koncepcjami projektowymi Python zawiera również elementy takie, jak moduły czy programowanie zorientowane obiektywo, które promują możliwość ponownego użycia kodu. A ponieważ Python skupia się na jakości, podobnie robią programiści piszący w tym języku.

Wydajność programistów

W czasie boomu internetowego w latach dziewięćdziesiątych ubiegłego wieku trudno było znaleźć wystarczającą liczbę programistów, którzy mogliby implementować projekty informatyczne. Programiści mieli implementować systemy tak szybko, jak szybko rozwijał się Internet. Gdy tamten etap odszedł w przeszłość i nastąpiły czasy zastoju, recesji i masowych zwolnień, role się odwróciły. Programiści byli często proszeni o wykonywanie tych samych zadań z pomocą jeszcze mniejszej liczby osób.

W obu scenariuszach Python świetnie się sprawdzał jako narzędzie, które pozwala programistom mniejszym wysiłkiem osiągnąć więcej. Python celowo zoptymalizowany jest pod kątem *szynkości programowania* — jego prosta składnia, dynamiczne typy, brak komplikacji i wbudowany zestaw narzędzi pozwalają tworzyć programy w ułamku czasu, jaki potrzebny byłby do uzyskania tego samego efektu za pomocą innych metod. Efekt jest taki, że wydajność programisty piszącego w Pythonie zazwyczaj jest o wiele wyższa niż wydajność osoby piszącej w innych językach programowania. To dobra wiadomość zarówno na lepsze, jak i na gorsze czasy, a także na każdy okres pomiędzy nimi, w jakim może się znajdować przemysł informatyczny.

Czy Python jest językiem skryptowym?

Python jest językiem programowania ogólnego przeznaczenia, który często wykorzystywany jest do tworzenia skryptów. Często nazywa się go *zorientowanym obiektywo skryptowym językiem programowania* — w tej definicji łączy się obsługę programowania zorientowanego obiektywo z przeznaczeniem do tworzenia skryptów. Jeżeli miałbym to podsumować jednym zdaniem, powiedziałbym, że Python jest prawdopodobnie lepiej znany jako *uniwersalny język programowania, który łączy paradygmaty proceduralne, funkcjonalne i zorientowane obiektywo* — jest to stwierdzenie, które dobrze oddaje bogactwo i aktualne możliwości Pythona.

Nie zmienia to jednak w niczym faktu, że określenie „język skryptowy” zdaje się ciągle trzymać się Pythona jak rzep psiego ogona, być może dla podkreślenia większego wysiłku, jaki programista musi włożyć w osiągnięcie podobnych rezultatów przy użyciu innych narzędzi. Na przykład, mówiąc o plikach kodu Pythona ludzie często używają słowa „skrypt” zamiast „program”. Zgodnie z tą tradycją, w naszej książce oba terminy używane są wymiennie, z tym, że słowo „skrypt” jest preferowane w kontekście prostych plików, natomiast określenia „program” staramy się używać w odniesieniu do bardziej skomplikowanych aplikacji składających się z wielu plików.

Ponieważ pojęcie „język skryptowy” dla każdego może znaczyć co innego, niektóre osoby wolałyby, żeby w ogóle nie stosować go w kontekście Pythona. W praktyce określenie takie przywołuje zazwyczaj jedno z trzech bardzo różniących się od siebie skojarzeń:

Narzędzia powłoki

Czasami kiedy ludzie słyszą, że Python jest językiem skryptowym, sądzą, że oznacza to, iż jest narzędziem do tworzenia skryptów przeznaczonych dla systemu operacyjnego. Takie programy często uruchamiane są z poziomu wiersza poleceń i wykonują różne zadania, takie jak przetwarzanie plików tekstowych i uruchamianie innych programów.

Programy w Pythonie mogą spełniać takie role i robią to, jednak to tylko jedno z dziesiątek różnych zastosowań Pythona. Python nie jest nieco ulepszonym językiem skryptowym powłoki.

Język zarządzania

Dla innych użytkowników język skryptowy odnosi się do warstwy „spajającej”, wykorzystywanej do kontrolowania innych komponentów aplikacji i sterowania nimi. Programy w języku Python faktycznie są często wykorzystywane w kontekście większych aplikacji. Żeby na przykład przetestować jakieś urządzenie, programy w języku Python mogą wywoływać komponenty dające niskopoziomowy dostęp do tego urządzenia. I podobnie — programy mogą wykonywać kod napisany w Pythonie w strategicznych miejscach i momentach, by dostosować działanie produktu do wymagań użytkownika bez konieczności ponownego kompilowania i przesyłania kodu źródłowego całego systemu.

Prostota Pythona sprawia, że jest on naturalnym i elastycznym narzędziem do zarządzania. Z technicznego punktu widzenia jest to jednak tylko jedna z wielu ról, w jakich Python może występować. Wielu programistów (być może nawet większość z nich) tworzy w tym języku samodzielne skrypty, nie wiedząc o istnieniu żadnych zintegrowanych komponentów. Python nie jest tylko językiem służącym do zarządzania innymi elementami.

Łatwość użycia

Chyba najbardziej oczywistym rozwinięciem pojęcia „język skryptowy” jest odniesienie do prostego języka wykorzystywanego do nieskomplikowanych, szybkich do wykonania zadań programistycznych. Takie rozumienie tego terminu szczególnie dobrze odnosi się do języka Python, który pozwala na znacznie szybsze tworzenie kodu niż za pomocą języków kompilowanych, takich jak C++. Szybki cykl tworzenia oprogramowania zachęca do poszukiwania najlepszych rozwiązań i ciągłego ulepszania istniejących.

Nie daj się jednak zwieść pozoram — Python nie służy tylko do wykonywania prostych zadań. Lepiej będzie powiedzieć, że to sam język upraszcza wiele zadań dzięki swojej elastyczności i łatwości użycia. Python ma prosty zestaw funkcji, ale pozwala na łatwe tworzenie złożonych programów i rozbudowywanie ich w miarę potrzeb. Z tego powodu wykorzystywany jest zarówno w krótkich, taktycznych zadaniach, jak i w długoterminowych, skomplikowanych projektach programistycznych.

Czy zatem Python jest językiem skryptowym? Zależy, kogo o to zapytać. Wydaje się, że określenia „skryptowy” najlepiej będzie używać w odniesieniu do szybkiego, elastycznego trybu programowania oferowanego przez ten język, a nie do konkretnej dziedziny jego zastosowań.

Jakie są wady języka Python?

Po dwudziestu jeden latach używania Pythona, pisania o nim przez osiemnaście lat i uczenia go przez lat szesnaście, zauważylem, że jedną znaczącą wadą tego języka jest fakt, że prędkość działania programów w nim napisanych nie zawsze może być porównywalna z prędkością języków niskopoziomowych i kompilowanych, takich jak C czy C++. Choć w dzisiejszych czasach zdarza się to już stosunkowo rzadko, w przypadku niektórych zadań możesz być zmuszony do „zblżenia się do sprzętu” i użycia języków niższego poziomu, które pozwalają na bardziej bezpośrednie odwoływanie się do architektury urządzeń.

O koncepcjach wdrożenia pomówimy w kolejnych rozdziałach książki. W skrócie mówiąc, dzisiejsze standardowe implementacje Pythona kompilują (przekładają) instrukcje z kodu źródłowego na format pośredni nazywany *kodem bajtowym* (ang. *byte code*), a następnie interpretują kod bajtowy. Kod bajtowy zapewnia przenośność aplikacji, ponieważ jest to format niezależny od platformy. Ponieważ jednak kod w Pythonie nie jest kompilowany do poziomu binarnego kodu maszynowego (na przykład do postaci zestawów instrukcji dla procesorów

firmy Intel), niektóre programy napisane w języku Python będą działały wolniej niż aplikacje napisane w językach w pełni kompilowanych, takich jak C. Przykładowo, *PyPy*, czyli implementacja języka Python napisana w języku RPython (ang. *Restricted Python*), pozwala na tworzenie programów, które można skompilować do kodu bajtowego Javy, CLR czy języka C, co w efekcie daje zwiększenie szybkości działania programu od kilkunastu do nawet kilkudziesięciu razy w porównaniu do „klasycznego” Pythona, ale jednak *PyPy* to już zupełnie osobne, alternatywne rozwiązanie.

To, czy różnica w szybkości wykonywania będzie miała *jakieś znaczenie*, zależy od typu programów, jakie będziemy pisać. Python wiele razy był optymalizowany, a kod napisany w tym języku sam z siebie działa wystarczająco szybko dla większości zastosowań. Co więcej, w przypadku większości poważnych zadań w skryptach napisanych w języku Python, takich jak przetwarzanie plików czy tworzenie graficznych interfejsów użytkownika (GUI), program i tak działa z prędkością programu w języku C, ponieważ takie zadania są wewnątrz interpretera Pythona natychmiast przekazywane do skompilowanego kodu napisanego w C. Co więcej, łatwość, z jaką możemy tworzyć programy w języku Python, jest zazwyczaj o wiele ważniejsza od nieco mniejszej szybkości działania, w szczególności na współczesnych komputerach.

Jednak nawet przy dzisiejszych możliwościach procesorów istnieją dziedziny, w których optymalna szybkość wykonywania jest istotna. W przypadku złożonych obliczeń numerycznych czy animacji z pewnością nie zaszkodzi, aby przynajmniej najważniejsze komponenty przetwarzające liczby działały z taką prędkością, jak programy napisane w języku C (lub nawet większą). Oczywiście w takich zastosowaniach nadal można wykorzystywać Pythona — wystarczy tylko wyodrębnić części aplikacji wymagające optymalnej szybkości i utworzyć z nich *skompilowane rozszerzenia*, które następnie można połączyć w całość za pomocą skryptów napisanych w języku Python.

W niniejszej książce nie będziemy jednak zbyt wiele mówić o takich rozszerzeniach, aczkolwiek jest to jeden z przypadków zastosowania Pythona jako języka sterującego działaniem aplikacji. Doskonałym przykładem takiego rozwiązania jest biblioteka *NumPy*, wspomagająca przeprowadzanie obliczeń numerycznych. Dzięki połączeniu skompilowanych i zoptymalizowanych rozszerzeń numerycznych z językiem Python, NumPy sprawia, że Python staje się wydajnym i łatwym w użyciu narzędziem do programowania obliczeń numerycznych. W razie potrzeby takie rozszerzenia dostarczają bardzo przydatnych mechanizmów optymalizacyjnych.

Inne kompromisy w języku Python: imponderabilia

Wspomniałem już, że szybkość działania programów jest jednym poważnym mankamentem języka Python, a przynajmniej tak właśnie uważa większość użytkowników Pythona, zwłaszcza tych nowych. Większość ludzi uważa, że Python jest łatwy do nauczenia się i przyjemny w użyciu, szczególnie w porównaniu z innymi językami programowania, takimi jak Java, C# i C++. Aby jednak wszystko było jasne, powinieneś zwrócić uwagę na kilka innych kompromisów istniejących w świecie Pythona, które obserwowałem przez dwie dekady pracy z tym językiem — zarówno jako wykładowca, jak i programista.

Jako wykładowcy nie raz zdarzało mi się narzekać na *ogromne tempo*, w jakim przez te wszystkie lata rozwijał się język Python i jego biblioteki. Działo się tak po części dlatego, że zarówno wykładowcy, jak i autorzy książek znajdują się niejako na pierwszej linii frontu takich zmian — moim zadaniem było wszak nauczanie tego języka pomimo jego ciągłej ewolucji, co czasami było i jest zadaniem przypominającym próbę zapanowania nad stadem niesfornych kotów! Jest to jednak powszechnie znany problem. Jak zobaczymy w tej książce, sztandarowe motto języka Python — „zachowaj prostotę” — jest dziś często podporządkowywane potrzebom wdrażania coraz bardziej wyrafinowanych rozwiązań przy coraz bardziej powiększającym się zakresie materiału do opanowania dla nowych adeptów programowania w tym pięknym języku. Rozmiar tej książki jest pośrednim dowodem narastania tego trendu.

Z drugiej strony, w większości rozwiązań Python jest nadal znacznie łatwiejszy niż jego alternatywy; inaczej mówiąc, jest tylko tak złożony, jak tego wymagają zadania, które są

przy jego pomocy realizowane. Mimo to, jego ogólna spójność i otwarty charakter nadal pozostają przekonywującymi cechami dla większości użytkowników. Co więcej, nie każdy z nich musi być na bieżąco z najnowszymi implementacjami, na co wyraźnie wskazuje aktualna popularność języka Python w wersji 2.x.

Jako *programista*, czasami kwestionuję również kompromisy związane z podejściem typu „baterie są dołączone do zestawu”, prezentowanym przez deweloperów rozwijających Pythona. Nacisk, jaki kładzie się obecnie na wstępnie przygotowane narzędzia, może implikować nowe zależności (co jeżeli bateria, której używasz, zostanie zmieniona, jest uszkodzona lub po prostu przestarzała?) i zachęcać do stosowania nietypowych, dedykowanych rozwiązań, nie zawsze zgodnych z ogólną filozofią języka, ale mogących lepiej służyć użytkownikom na dłuższą metę (jak możesz ocenić działanie narzędzia lub używać go, jeżeli nie rozumiesz jego przeznaczenia i sposobu działania?). Przykłady obu tych problemów pokażemy w tej książce. Dla typowych użytkowników, a szczególnie dla hobbystów i początkujących, takie podejście do modelu zestawów narzędzi jest jednym z głównych atutów języka Python. Nie powinieneś być jednak zaskoczony, jeżeli po pewnym czasie Twoje programy przerosną dostępne wcześniej, predefiniowane narzędzia, zwłaszcza kiedy będziesz korzystać z wiedzy, którą ta książka ma na celu przekazać.

Parafrując dobrze znane przysłowie: daj ludziom narzędzie, a będą kodować przez jeden dzień; naucz ich, jak budować narzędzia, a będą pisać swoje programy przez całe życie. Nasza książka bardziej skłania się ku temu drugiemu rozwiązaniu.

Jak już wspominaliśmy w innym miejscu tego rozdziału, zarówno sam język Python, jak i jego model zestawu narzędzi, są podatne na wady wspólne dla projektów *open source* w ogóle — potencjalną przewagę *osobistych preferencji* kilku deweloperów nad preferencjami wielu użytkowników i okazjonalne pojawianie się *anarchii*, a nawet *elitarystyzmu* — choć takie zjawiska są zazwyczaj najbardziej bolesne w czasie pojawiania się nowych wersji oprogramowania.

Wróćmy do niektórych z tych kompromisów na końcu książki, po tym, jak nauczysz się Pythona wystarczająco dobrze, abyś mógł wyciągać swoje własne wnioski. Jako rozwiązanie typu *open source* to, czym „jest” Python, zależy w dużej mierze od jego użytkowników — w końcu Python jest dziś bardziej popularny niż kiedykolwiek wcześniej, a tempo jego rozwoju nie wykazuje żadnych oznak osłabienia. Dla wielu użytkowników może to być znacznie bardziej wymowne i przekonujące niż indywidualne opinie poszczególnych ludzi.

Kto dzisiaj używa Pythona?

W chwili pisania niniejszej książki można było oszacować, że na całym świecie jest obecnie około miliona użytkowników języka Pythona. Ta liczba jest oparta na różnych statystykach, na przykład liczbie pobrań ze strony Pythona, danych dotyczących sieci czy ankietach przeprowadzanych wśród programistów. Ponieważ Python jest produktem typu *open source* (o otwartym kodzie źródłowym), trudno jest dokonać dokładniejszych obliczeń, ponieważ nie można zliczać liczb wykupionych licencji. Co więcej, Python jest częścią dystrybucji systemu Linux, systemów operacyjnych komputerów Macintosh, a także wielu innych produktów czy urządzeń, co jeszcze bardziej utrudnia dokładniejsze oszacowanie liczby jego użytkowników.

Ogólnie rzecz biorąc, Python cieszy się sporą bazą użytkowników; wokół tego języka skupiona jest też bardzo aktywna społeczność programistów. Ogólnie uważa się, że jest on w *pierwszej piątce lub przynajmniej dziesiątce* najpopularniejszych języków programowania na świecie (dokładne rankingi różnią się w zależności od źródła i daty). Ponieważ język ten istnieje od ponad dwudziestu lat i jest w szerokim użyciu, jest bardzo stabilny i ma duże możliwości.

Oprócz używania przez indywidualne osoby język Python jest również stosowany w wielu produktach generujących przychody rozmaitym firmom. Na przykład:

- Firma *Google* intensywnie wykorzystuje Pythona w swoich wyszukiwarkach.
- Popularny serwis służący do udostępniania filmów i klipów wideo *YouTube* jest w większości napisany w języku Python.
- Firma *Dropbox*, udostępniająca usługi przechowywania danych w chmurze, pisze swoje oprogramowanie — zarówno po stronie serwera, jak i klienta — głównie w języku Python.
- Jednoprzyłtkowy komputer *Raspberry Pi* wykorzystuje Pythona w celach edukacyjnych.
- Python jest intensywnie wykorzystywany w *EVE Online*, grze typu MMOG (ang. *Massively Multiplayer Online Game*) firmy CCP Games.
- *BitTorrent*, czyli popularny system dzielenia się plikami w systemie *peer-to-peer*, jest programem napisanym w języku Python.
- Firmy *Industrial Light & Magic*, *Pixar* i wiele innych inne wykorzystują Pythona w tworzeniu filmów animowanych.
- Firma *ESRI* (ang. *Environmental Systems Research Institute*) używa języka Python jako narzędzia służącego do dostosowania swoich systemów informacji geograficznej (ang. *GIS — Geographic Information System*) do potrzeb użytkowników końcowych.
- Popularna platforma wspomagająca tworzenie aplikacji webowych *App Engine* firmy Google wykorzystuje Pythona w roli języka aplikacji.
- Oprogramowanie serwera poczty elektronicznej *IronPort* wykorzystuje ponad milion wierszy kodu napisanego w języku Python.
- *Maya*, potężny zintegrowany system modelowania i animacji 3D, udostępnia interfejs API dla skryptów w języku Python.
- Agencja *NSA* (ang. *National Security Agency*) wykorzystuje język Python w kryptografii oraz analizach wywiadowczych.
- Firma *iRobot* wykorzystuje Pythona do tworzenia oprogramowania swoich robotów przeznaczonych zarówno na rynek cywilny, jak i do zastosowań militarnych.
- Modyfikacje gry *Civilization IV*, pozwalające m.in. na tworzenie zdarzeń, są w całości pisane w języku Python.
- Interfejs użytkownika oraz modele aktywności projektu OLPC (ang. *One Laptop Per Child*) są tworzone w języku Python.
- W dokumentacji usług *Netflix* oraz *Yelp* można znaleźć wiele informacji o roli języka Python w oprogramowaniu ich systemów.
- Firmy *Intel*, *Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm* oraz *IBM* wykorzystują Pythona do testowania urządzeń sprzętowych.
- Firmy *JPMorgan Chase*, *UBS*, *Getco* i *Citadel* używają Pythona podczas tworzenia prognoz finansowych.
- Instytucje takie jak *NASA*, *Los Alamos*, *Fermilab*, *JPL* i inne, wykorzystują Pythona do zadań programistycznych w różnych dziedzinach nauki.

I tak dalej — chociaż zamieszczone powyżej zestawienie jest dosyć reprezentatywne, to z oczywistych względów przedstawienie pełnej listy zastosowań wykracza daleko poza zakres tej książki, a przy tym możemy mieć niemal pewność, że taka lista w miarę upływu czasu będzie się bardzo szybko rozbudowywać. Aby sprawdzić aktualne przykłady zastosowań języka Python w aplikacjach i oprogramowaniu, możesz zatrzymać się na stronie projektu Python, poczytać w Wikipedii lub po prostu zadać kilka prostych zapytań swojej ulubionej wyszukiwarce internetowej:

- Historie sukcesu: <http://www.python.org/about/success>
- Obszary zastosowań: <http://www.python.org/about/apps>
- Opinie użytkowników: <http://www.python.org/about/quotes>
- Strona Wikipedii: http://en.wikipedia.org/wiki/List_of_Python_software

Prawdopodobnie jedynym wspólnym wątkiem łączącym firmy używające dziś Pythona jest to, że język ten jest popularny na całym świecie (w kontekście zakresu zastosowań). Python jest językiem ogólnego zastosowania, dzięki czemu można go wykorzystywać w prawie wszystkich dziedzinach. Można właściwie powiedzieć, że Python wykorzystywany jest przez prawie każdą znaczącą organizację tworzącą oprogramowanie, czy to w przypadku krótkoterminowych zadań

taktycznych, takich jak testowanie i zarządzanie, czy też w przypadku długoterminowego rozwoju produktów strategicznych. Język ten znakomicie sprawdza się w obu scenariuszach.

Co mogę zrobić za pomocą Pythona?

Python jest nie tylko dobrze zaprojektowanym językiem programowania — to także język przydatny w wykonywaniu rzeczywistych zadań, takich, z jakimi programiści spotykają się na co dzień. Jest używany w wielu różnych zastosowaniach jako narzędzie do tworzenia zarówno skryptów dla innych komponentów, jak i samodzielnych programów i aplikacji. W praktyce, będąc językiem ogólnego przeznaczenia Python może być wykorzystywany właściwie wszędzie — można go zastosować w dowolnej dziedzinie, począwszy od tworzenia witryn internetowych, poprzez programowanie gier, aż do sterowania robotami i statkami kosmicznymi.

Najczęściej jednak dziedziny, w jakich wykorzystywany jest Python, dzielą się na kilka szerszych kategorii. Poniżej opisano kilka najczęściej spotykanych zastosowań języka Python wraz z narzędziami wykorzystywanymi w danej sytuacji. Nie będziemy w stanie omówić tych narzędzi bardziej szczegółowo, dlatego osoby zainteresowane tymi zagadnieniami odsyłam na oficjalną stronę projektu Python lub do innych źródeł.

Programowanie systemowe

Wbudowane interfejsy do usług systemów operacyjnych sprawiają, że Python idealnie nadaje się do pisania przenośnych i łatwych w aktualizowaniu narzędzi służących do zarządzania systemami (czasami nazywanych *narzędziami powłoki* — ang. *shell tools*). Programy napisane w tym języku mogą służyć na przykład do przeszukiwania plików i drzew katalogów, uruchamiania innych programów czy wykonywania przetwarzania równoległego za pomocą procesów i wątków.

Standardowa biblioteka Pythona zawiera wiązania POSIX i obsługę wszystkich najczęściej spotykanych elementów systemu operacyjnego — zmiennych środowiskowych, plików, gniazd, potoków, procesów, wielu wątków, dopasowywania wzorców wyrażeń regularnych, argumentów przekazywanych z poziomu wiersza poleceń, standardowych strumieni danych, programów uruchamianych z wiersza poleceń, rozszerzeń nazw plików, plików ZIP, XML, JSON, CSV i wielu innych. Dodatkowo większość interfejsów systemowych Pythona zaprojektowano tak, aby mogły być przenośne. Przykładowo, skrypt kopiący drzewa katalogów zazwyczaj będzie działał tak samo na wszystkich najważniejszych platformach bez konieczności wprowadzania jakichkolwiek modyfikacji. Interpreter *Stackless Python* (zobacz rozdział 2.), który wykorzystywany jest w grze *EVE Online*, oferuje zaawansowane rozwiązania przetwarzania wieloprocesowego (ang. *multiprocessing*).

Graficzne interfejsy użytkownika (GUI)

Prostota oraz szybkość programowania w Pythonie sprawiają, że język ten często wykorzystywany jest w programowaniu graficznych interfejsów użytkownika GUI (ang. *graphical user interface*). W standardowej implementacji języka Python znajdziemy standardowy, zorientowany obiektywnie interfejs API o nazwie *tkinter* (w wersjach Python 2.x nosi nazwę *Tkinter*), pozwalający na tworzenie w programach napisanych w języku Python prostych interfejsów GUI, w pełni komponujących się z wyglądem danego systemu operacyjnego. Graficzne interfejsy użytkownika Python/tkinter działają bez konieczności wprowadzania jakichkolwiek zmian w systemach Microsoft Windows, X Windows (Unix oraz Linux), a także macOS (zarówno Classic, jak i OS X). Darmowy pakiet rozszerzeń, *PMW*, dodaje do interfejsu tkinter zaawansowane widgety. Oprócz tego istnieje również oparty na bibliotece

C++ pakiet o nazwie *wxPython*, który oferuje alternatywny zbiór narzędzi służących do tworzenia graficznych interfejsów użytkownika wykorzystywanych na różnych platformach.

Zestawy narzędzi wyższego poziomu, takich jak *Dabo*, zbudowane są na bazie interfejsów API modułów takich jak *wxPython* oraz *tkinter*. Za pomocą odpowiednich bibliotek języka Python można również korzystać z innych interfejsów GUI, takich jak *Qt* (z wykorzystaniem biblioteki *PyQT*), *GTK* (biblioteka *PyGTK*), *MFC* (biblioteka *PyWin32*), *.NET* (biblioteka *IronPython*) czy *Swing* (biblioteki *JPype* lub *Jython*; *Jython* to opisana w rozdziale 2. implementacja języka skryptowego Python napisana w języku Java i zintegrowana z platformą Java). Dla aplikacji działających w przeglądarkach lub mających małe wymagania w zakresie interfejsów użytkownika, zarówno *Jython*, jak i opisane poniżej platformy webowe wykorzystujące Pythona oraz skrypty CGI po stronie serwera udostępniają dalsze opcje.

Skrypty internetowe

Python zawiera standardowe moduły internetowe, które pozwalają programom napisanym w tym języku na wykonywanie różnorodnych zadań sieciowych zarówno w trybie klienta, jak i serwera. Skrypty mogą się komunikować za pośrednictwem gniazd sieciowych, mogą pobierać informacje z formularzy przesyłanych do skryptów CGI po stronie serwera, przesyłać pliki za pomocą protokołu FTP, parsować, generować i przetwarzanie pliki w formatach XML i JSON, wysyłać, otrzymywać, tworzyć i przetwarzanie wiadomości e-mail, pobierać całe strony internetowe z podanych adresów URL, przetwarzanie kod HTML pobranych stron, komunikować się za pośrednictwem protokołów XML-RPC, SOAP czy Telnet i wielu innych. Odpowiednie biblioteki języka Python sprawiają, że takie zadania stają się zaskakująco proste.

Dodatkowo w internecie dostępnych jest wiele zestawów narzędzi tworzonych przez niezależnych programistów, dzięki którym programowanie webowe w Pythonie staje się jeszcze łatwiejsze. Na przykład, biblioteka *HTMLGen* wspomaga generowanie dokumentów HTML za pomocą skryptów języka Python opartych na klasach, a pakiet *mod_python* uruchamia Pythona na serwerze Apache i włącza obsługę szablonów po stronie serwera za pomocą Python Server Pages. Z kolei pakiet *Jython* umożliwia bezproblemową integrację Pythona i Javy, a także obsługuje tworzenie appletów po stronie serwera, które działają na kliencie.

Istnieją również pełne pakiety wspomagające tworzenie stron internetowych (ang. *web development framework*), takie jak *Django*, *TurboGears*, *web2py*, *Pylons*, *Zope* czy *WebWare*, które umożliwiają szybkie tworzenie pełnowymiarowych i wysokiej jakości stron internetowych za pomocą Pythona. Wiele z nich udostępnia opcje takie, jak obiektywo-relacyjne narzędzia odwzorowujące, wzorce MVC (Model-View--Controller — model-widok-kontroler), skrypty i szablony po stronie serwera, a także obsługę Ajaksa, łącząc je w kompletnie rozwiązania służące do profesjonalnego programowania webowego.

Niedawno Python został rozbudowany o możliwości tworzenia rozbudowanych aplikacji internetowych (RIA — ang. *Rich Internet Application*) z wykorzystaniem narzędzi takich jak *Silverlight* w *IronPython*, *pyjs* (znanego również pod nazwą *pyjamas*), kompilatorów Python-to-JavaScript, frameworka AJAX i różnych zestawów widgetów. Nowoczesny Python posiada również możliwości wykorzystywania technologii chmurowych dzięki zastosowaniu platformy *App Engine* i innych elementów opisanym w sekcji „Programowanie bazodanowe” poniżej. Gdzie „idzie” sieć, tam Python szybko podąża za nią.

Integracja komponentów

Integrację komponentów omówiliśmy już wcześniej, kiedy opisywaliśmy Pythona jako język zarządzania. Możliwość rozszerzania Pythona za pomocą języków C i C++, a także osadzania go w kodzie napisanym w tych językach sprawia, że jest on niezwykle przydatnym społkiem służącym do tworzenia skryptów sterujących zachowaniem innych podsystemów i komponentów aplikacji. Integracja biblioteki języka C z Pythonem pozwala na przykład

Pythonowi na testowanie i uruchamianie komponentów tej biblioteki. Osadzenie Pythona w takim produkcie pozwala na wprowadzanie zmian na miejscu, bez konieczności ponownej komplikacji całego pakietu (czy przesyłania jego kodu źródłowego).

Narzędzia, takie jak generatory kodu *SWIG* i *SIP*, mogą zautomatyzować większość potrzebnych do połączenia skompilowanych komponentów z Pythonem w celu użycia w skryptach, a język *Cython* umożliwia programistom mieszanie kodu Pythona i kodu podobnego do języka C. Większe frameworki, takie jak obsługa COM w systemie Windows, implementacja Jython oparta na *Javie*, oparta na platformie .NET implementacja IronPython, udostępniają alternatywne sposoby tworzenia skryptów dla poszczególnych komponentów aplikacji. W systemie Windows skrypty napisane w Pythonie można na przykład wykorzystać w połączeniu z programami Microsoft Word czy Excel, technologią *Silverlight* i wieloma innymi komponentami.

Programowanie bazodanowe

Python posiada interfejsy do wszystkich najpopularniejszych relacyjnych baz danych, takich jak Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite. W świecie Pythona został również zdefiniowany *uniwersalny interfejs API dla baz danych*, pozwalający na uzyskanie do dostępu do systemów baz danych SQL z poziomu skryptów napisanych w języku Python; wygląda on tak samo dla wielu różnych systemów baz danych. Ponieważ taki uniwersalny interfejs API jest zaimplementowany w rozwiązańach bazodanowych wielu różnych producentów, skrypt napisany dla darmowej bazy danych MySQL będzie w zasadzie działał bez większych zmian w innych systemach baz danych (na przykład Oracle). Wszystko, co musisz zrobić, to zastąpić podstawowy interfejs dostawcy. Wbudowany silnik *SQLite* bazy danych SQL jest już standardową częścią samego Pythona od dobrych trzech lat i obsługuje zarówno prototypowanie, jak i podstawowe zapotrzebowanie aplikacji na przechowywanie danych.

W dziale baz typu NoSQL możemy korzystać z dostępnego w języku Python standardowego modułu *pickle*, który zapewnia prosty system *utrwalania obiektów* (ang. *object persistence*), pozwalający na łatwe zapisywanie całych obiektów Pythona do plików lub obiektów podobnych do plików, a następnie przywracanie ich do oryginalnej postaci. W sieci znajdziesz także systemy bazodanowe typu *open source ZODB* i *Durus*, które dostarczają kompletne, zorientowane obiektywne systemy baz danych dla skryptów Pythona; inne, takie jak *SQLObject* i *SQLAlchemy*, wprowadzają mapowanie obiektywno-relacyjne (ang. *ORM — Object-Relational Mapping*), które odwzorowuje model klas Pythona na tabele relacyjne. Innym przykładem może być *PyMongo*, czyli interfejs do bazy *MongoDB*, bardzo efektywnej bazy danych typu NoSQL o otwartym kodzie źródłowym, przechowującej dane w strukturach bardzo podobnych do własnych list i słowników Pythona, które mogą być analizowane i tworzone za pomocą własnej, standardowej biblioteki *json* języka Python.

Jeszcze inne systemy oferują bardziej wyspecjalizowane sposoby przechowywania danych; przykładem może być magazyn danych *App Engine* firmy Google, który modeluje dane za pomocą klas Pythona i zapewnia dużą skalowalność całego rozwiązania; oprócz tego dostępnych jest wiele innych opcji przechowywania danych w chmurze, takich jak *Azure*, *PiCloud*, *OpenStack* czy *Stackato*.

Szybkie prototypowanie

Dla programów w języku Python, komponenty napisane w Pythonie i języku C wyglądają tak samo. Z tego powodu możliwe jest początkowe prototypowanie w Pythonie i późniejsze przenoszenie wybranych komponentów do języka kompilowanego, takiego jak C czy C++. W przeciwnieństwie do niektórych narzędzi do prototypowania, Python nie wymaga całkowitego przepisania komponentu po jego ustabilizowaniu. Komponenty systemu, które nie potrzebują wydajności takiej jak w języku C++, mogą pozostać napisane w języku Python ze względu na łatwość modyfikowania i używania.

Programowanie numeryczne i naukowe

Python jest również z powodzeniem wykorzystywany w programowaniu numerycznym, czyli dziedzinie, która tradycyjnie nie była uważana za domenę języków skryptowych, ale która stała się jednym z najbardziej interesujących zastosowań Pythona. Wspomniana wcześniej biblioteka *NumPy*, wspomagająca przeprowadzanie obliczeń numerycznych, zawiera szereg zaawansowanych narzędzi, takich jak obiekty tablic czy interfejsy do standardowych bibliotek matematycznych. Integrując Pythona z szybko działającymi procedurami numerycznymi zakodowanymi w języku komplikowanym, biblioteka *NumPy* sprawia, że Python staje się zaawansowanym, a przy tym łatwym w użyciu narzędziem do programowania numerycznego, które często jest w stanie zastąpić istniejący kod napisany w tradycyjnych językach komplikowanych, takich jak FORTRAN czy C++.

Dodatkowe narzędzia numeryczne dla Pythona obsługują między innymi animacje, wizualizacje trójwymiarowe i przetwarzanie równoległe. Na przykład, popularne biblioteki *SciPy* oraz *ScientificPython*, bazujące na bibliotece *NumPy*, udostępniają dodatkowe zestawy narzędzi służących do zastosowań naukowych. Implementacja PyPy języka Python (która została omówiona w rozdziale 2.) zyskała również na znaczeniu w zastosowaniach numerycznych, częściowo dlatego, że napisane w PyPy złożone algorytmy stosowane w obliczeniach numerycznych zazwyczaj działają od kilkunastu do kilkudziesięciu razy szybciej w porównaniu do klasycznego Pythona.

I dalej: gry, przetwarzanie obrazu, wyszukiwanie danych, robotyka, Excel...

Python znajduje zastosowanie w zdecydowanie większej liczbie dziedzin, niż mogliśmy tutaj wymienić. Poniżej przedstawiamy kilka przykładowych domen, w których znajdziesz narzędzia korzystające z tego języka:

- Programowanie gier oraz multimediiów, dzięki zastosowaniu bibliotek *pygame*, *cgkit*, *pyglet*, *PySoy*, *Panda3D* i innych.
- Komunikacja przez porty szeregowe w systemach Windows, Linux i innych, realizowana dzięki bibliotece *PySerial*.
- Przetwarzanie obrazu z wykorzystaniem biblioteki *PIL* i jej nowej odnogi *Pillow*, czy narzędzi takich jak *PyOpenGL*, *Blender*, *Maya* i innych.
- Sterowanie robotami za pomocą zestawu narzędzi *PyRo*.
- Analiza i przetwarzanie języków naturalnych dzięki pakietowi *NLTK* (ang. *Natural Language Toolkit*).
- Oprogramowanie oprzyrządowania w komputerach opartych na *Raspberry Pi* oraz *Arduino*.
- Urządzenia mobilne z implementacjami Pythona na platformach Google *Android* i Apple *iOS*.
- Funkcje arkusza kalkulacyjnego Excel i programowanie makr z wykorzystaniem dodatków *PyXLL* lub *DataNitro*.
- Przetwarzanie zawartości i metadanych plików multimedialnych z użyciem bibliotek *PyMedia*, *ID3*, *PIL/Pillow* i wielu innych.
- Rozwiązania sztucznej inteligencji z wykorzystaniem biblioteki sieci neuronowych *PyBrain* i zestawu narzędzi do uczenia maszynowego *Milk*.
- Tworzenie systemów ekspertowych z wykorzystaniem bibliotek *PyCLIPS*, *Pyke*, *Pyrolog* i *PyDatalog*.
- Monitorowanie sieci za pomocą pakietu *zenoss*, napisanego w dużej części w języku Python.
- Projektowanie i modelowanie z użyciem pakietów *PythonCAD*, *PythonOCC*, *FreeCAD* i innych.

- Przetwarzanie i generowanie dokumentów za pomocą pakietów *ReportLab*, *Sphinx*, *Cheetah*, *PyPDF* i innych.
- Wizualizacja danych za pomocą pakietów i bibliotek *Mayavi*, *matplotlib*, *VTK*, *VPython* i innych.
- Przetwarzanie dokumentów XML za pomocą biblioteki *xml*, modułu *xmllib* oraz innych rozszerzeń.
- Przetwarzanie plików JSON i CSV za pomocą bibliotek *json* i *csv*.
- Analiza danych z wykorzystaniem framework'a *Orange*, pakietów *Pattern*, *Scrapy* oraz innych rozwiązań.

Dzięki pakietowi *PySol* możemy nawet ułożyć sobie pasjansa! Oczywiście, możesz również używać języka Python w wielu innych, mniej popularnych zastosowaniach, wykonywać przy jego pomocy codzienne zadania administratora systemu, przetwarzać pocztę elektroniczną, zarządzać bibliotekami dokumentów i multimediiów, i tak dalej. Wiele przykładów takich zastosowań możesz znaleźć na stronie internetowej projektu *PyPI*; możesz także użyć wyszukiwarki sieciowej Google lub po prostu poszukać na stronie <http://www.python.org>.

W zdecydowanej większości przedstawionych zastosowań rolą języka Python jest integracja komponentów aplikacji i systemu w jedną spójną całość. Używanie Pythona jako frontendu bibliotek komponentów napisanych w językach kompilowanych (takich, jak C) sprawia, że Python znakomicie ułatwia pisanie skryptów dedykowanych do wielu różnych zastosowań. Python jest szeroko stosowany jako język ogólnego przeznaczenia wspomagający integrację różnych komponentów aplikacji.

Jak Python jest rozwijany i wspierany?

Jako popularny system typu *open source* Python cieszy się dużą i aktywną społecznością programistów, którzy reagują na zgłaszone problemy i proponują ulepszenia z szybkością, jaka może zrobić wrażenie na wielu twórcach programów komercyjnych. Programiści Pythona koordynują pracę online za pomocą systemu kontroli wersji kodu źródłowego (ang. *source-control system*). Wszelkie modyfikacje są opracowywane w oparciu o formalny protokół *PEP* (ang. *Python Enhancement Proposal* — „Propozycje ulepszenia Pythona”) lub inne, podobne standardy i muszą zawierać rozszerzenia pozwalające na przeprowadzanie testów regresyjnych. Tak naprawdę proces wprowadzania jakichkolwiek zmian do języka jest już podobnie złożony jak wprowadzanie zmian w produktach komercyjnych i nie ma wiele wspólnego z początkami Pythona, kiedy wystarczyło wysłać wiadomość z propozycjami zmian do jednego z deweloperów. Biorąc pod uwagę ogromną liczbę użytkowników, jest to jednak zmiana na lepsze.

Fundacja *PSF* (ang. *Python Software Foundation*), formalna grupa typu non-profit, organizuje konferencje i zajmuje się kwestiami własności intelektualnej. Na całym świecie odbywają się liczne konferencje na temat języka Python — największymi z nich są organizowana przez wydawnictwo O'Reilly OSCON (ang. *Open Source Convention*) oraz PyCon organizowana przez PSF (ang. *Python Software Foundation*). Pierwsza z nich poświęcona jest większej liczbie projektów *open source*, natomiast druga zajmuje się wyłącznie zagadnieniami związanymi z językiem Python i jego zastosowaniami, a liczba jej uczestników w ostatnich latach znaczowo rośnie. W konferencjach PyCon 2012 i 2013 wzięło udział po 2500 uczestników; w rzeczywistości na konferencji PyCon 2013 musiano ograniczyć limit miejsc do tego poziomu po niespodziewanej wyprzedaży wszystkich miejsc w 2012 r. (mimo tych ograniczeń konferencja cieszyła się szerokim zainteresowaniem obserwatorów, dziennikarzy i uczestników zarówno na gruncie technicznym, jak i nietechnicznym). We wcześniejszych latach liczba uczestników często podwajała się z roku na rok — przykładowo od 586 uczestników w 2007 r. do ponad 1000 w 2008, co świadczy o gwałtownie rosnącym zainteresowaniu Pythonem i może być szczególnie imponujące dla tych użytkowników, którzy pamiętają pierwsze konferencje poświęcone temu językowi, na których wszyscy uczestnicy i zaproszeni goście mogli się swobodnie pomieścić na lunchu przy kilku stolikach w jednej, niewielkiej restauracji).

Kompromisy związane z modelem open source

Nie zapominając o tym, co napisaliśmy powyżej, należy jednak zauważyc, że choć społeczność użytkowników i deweloperów Pythona rozwija się niezwykle dynamicznie, to jednak wiąże się to z nieuniknionymi kompromisami. Ścieżki rozwoju oprogramowanie typu *open source* często mogą się wydawać nieco chaotyczne, a nawet przypominać swego rodzaju *anarchię*, wskutek czego modyfikacje i nowe rozwiązania nie zawsze są tak płynnie wdrażane, jak mogłyby to wynikać z tego, o czym mówiliśmy przed chwilą. Niektóre zmiany w oprogramowaniu *open source* mogą być wprowadzane z pominięciem oficjalnych protokołów, a taki kod, jak każdy inny, jest podatny na błędy, które zdarzają się pomimo wysiłków deweloperów i starannego testowania nowego oprogramowania (na przykład Python w wersji 3.2.0 pojawił się z niepoprawnie działającą funkcją wprowadzania danych z konsoli w systemie Windows).

Co więcej, w projektach typu *open source* oczekiwania społeczności użytkowników co do sposobu funkcjonowania oprogramowania często muszą ustępować osobistym preferencjom zespołu deweloperów, którzy oczywiście mogą, ale nie muszą się zgadzać z opiniemi użytkowników — na dobrą sprawę użytkownicy takiego oprogramowania są w pewnym sensie na łasce tych deweloperów, którzy akurat mają wolny czas na wprowadzanie zmian i modyfikowanie kodu. Skutkiem takiego stanu rzeczy jest to, że ewolucja oprogramowania *open source* jest często napędzana przez nielicznych deweloperów, którzy siłą rzeczy narzucają całą rzeszy użytkowników swoją wizję rozwoju danego produktu.

W praktyce jednak takie kompromisy zdecydowanie bardziej wpływają na użytkowników starających się zawsze korzystać z jak najnowszych wersji danego oprogramowania, a nie na tych, którzy zadowalają się nieco starszymi, ale sprawdzonymi i stabilnymi wersjami, zarówno w wydaniach Python 3.x, jak i 2.x. Przykładowo, jeżeli nadal korzystasz z klasycznych klas dostępnych w Pythonie 2.x, to jesteś w dużej mierze odporny na prawdziwą *eksplozję* nowych funkcjonalności i zmian w klasach nowego typu, które zaczęły się pojawiać na przestrzeni ostatnich lat. Choć nowe klasy stały się de facto standardem w wydaniach Pythona serii 3.x (wraz z wieloma innymi zmianami), wielu użytkownikom serii 2.x nadal udaje się szczęśliwie omijać ten problem.

Jakie są techniczne mocne strony Pythona?

To oczywiście pytanie stawiane programistom i deweloperom. Dla osób, które nie mają doświadczenia w programowaniu, kilka kolejnych punktów może brzmieć niezrozumiale — nie ma się tym jednak co przejmować, wszystkie pojęcia zostaną omówione w dalszej części książki. Dla programistów niech to natomiast będzie szybkie wprowadzenie do niektórych najważniejszych możliwości technicznych Pythona.

Jest zorientowany obiektowo i funkcyjny

Python jest językiem zorientowanym obiektowo, od samych fundamentów. Jego model klas obsługuje zaawansowane koncepcje, takie jak polimorfizm, przeciążanie operatorów i dziedziczenie wielokrotne. W kontekście prostej składni oraz systemu typów Pythona aspekt zorientowania obiektowego jest wyjątkowo łatwy do implementacji. Co więcej, osoby niezaznajomione z tymi pojęciami szybko zauważą, że są one łatwiejsze do opanowania w Pythonie niż w dowolnym innym istniejącym języku programowania zorientowanym obiektowo.

Choć służy jako doskonałe narzędzie do strukturyzacji oraz ponownego wykorzystania kodu, zorientowanie obiektowe Pythona idealnie sprawdza się również w tworzeniu skryptów dla innych języków zorientowanych obiektowo, takich jak C++ czy Java. Za pomocą odpowiedniego

kodu spajającego programy napisane w Pythonie mogą służyć do tworzenia podklas dla klas zaimplementowanych w językach C++, Java czy C#.

Równie ważne jest to, że zorientowanie obiektowe jest w Pythonie *opcjonalne*. Wiele rzeczy można uzyskać bez konieczności wchodzenia na poziom guru programowania obiektowego. Podobnie jak C++, Python obsługuje zarówno tryb programowania proceduralnego, jak i zorientowanego obiektowo. Jego zorientowane obiektowo narzędzia mogą być stosowane, jeśli i kiedy pozwalają na to różne ograniczenia. Jest to szczególnie przydatne w fazach programowania taktycznego, które poprzedzają fazy projektowania.

Oprócz swoich paradygmatów *proceduralnych* (opartych na instrukcjach) i *obiektowych* (opartych na klasach), Python w ostatnich latach otrzymał obsługę *programowania funkcyjnego* — zestaw komponentów, który obejmuje generatory, wyrażenia listowe, domknięcia, mapy, dekoratory, anonimowe funkcje lambda i funkcje pierwnej klasy. Mogą one służyć zarówno jako uzupełnienie, jak i alternatywa dla narzędzi zorientowanych obiektowo.

Jest darmowy

Pythona można używać i dystrybuować zupełnie za darmo. Tak jak w przypadku innych produktów typu *open source*, takich jak Tcl, Perl, Linux czy Apache, kod źródłowy Pythona można pobrać za darmo z Internetu. Nie istnieją żadne ograniczenia dotyczące kopiowania go, osadzania we własnych rozwiązań czy dołączania Pythona do własnych produktów. Tak naprawdę można nawet sprzedawać Pythona, jeśli ktoś ma na to ochotę.

Nie należy jednak tego źle zrozumieć: „darmowy” nie znaczy wcale, że Python nie ma żadnego wsparcia technicznego. Wręcz przeciwnie — społeczność programistów i deweloperów Pythona odpowiada na pytania użytkowników z szybkością, jakiej może im pozazdrościć niejeden producent i sprzedawca oprogramowania komercyjnego. A ponieważ Python dostępny jest wraz z pełnym kodem źródłowym, daje to szerokie możliwości programistom, dzięki czemu z czasem powstała ogromna grupa specjalistów korzystających z tego języka. Choć analizowanie czy próba samodzielne modyfikowania oryginalnego kodu źródłowego Pythona nie należą może do ulubionych rozrywek każdego użytkownika, to jednak dobrze wiedzieć, że w razie potrzeby możesz tego spróbować. Nie jesteś w żaden sposób uzależniony od kaprysów wsparcia technicznego komercyjnego dostawcy, ponieważ ostateczna i najważniejsza dokumentacja każdego programu — jego *kod źródłowy* — jest w każdej chwili do Twojej dyspozycji.

Jak wspominaliśmy wcześniej, Python rozwijany jest przez społeczność, która w dużej mierze koordynuje swoje wysiłki za pomocą Internetu. W skład tej grupy wchodzi twórca Pythona — *Guido van Rossum*, oficjalnie noszący tytuł Benevolent Dictator for Life (BDFL — po polsku „dobrowolny, dożywotni dyktator”) — oraz tysiące innych osób wspierających. Wszelkie propozycje zmian w implementacji języka muszą przejść odpowiednią procedurę weryfikacyjną; muszą również być sprawdzone przez van Rossuma oraz innych deweloperów. Taka procedura zatwierdzania zmian sprawia, że w tej dziedzinie Python jest na szczęście bardziej konserwatywny niż wiele innych języków programowania. Choć podział na Pythona w wersji 3.x i 2.x w pewien sposób doprowadził do złamania tej tradycji, to nadal jej ogólne założenia są prawdziwe w każdej linii rozwoju tego języka.

Jest przenośny

Standardowa implementacja Pythona napisana jest w przenośnym języku ANSI C, dzięki czemu może być komplikowana i działa praktycznie na każdej ważniejszej platformie będącej obecnie w użytkowaniu. Programy napisane w Pythonie działają obecnie zarówno na urządzeniach mobilnych, jak i na superkomputerach. Python jest dostępny między innymi na platformach takich, jak:

- Linux oraz Unix,
- Microsoft Windows (we wszystkich współczesnych odmianach),

- macOS (zarówno OS X, jak i Classic),
- BeOS, OS/2, VMS oraz QNX,
- systemy czasu rzeczywistego, takie jak VxWorks,
- superkomputery Cray oraz duże systemy klasy mainframe (np. firmy IBM),
- urządzenia mobilne PDA działające pod kontrolą systemów Palm OS, PocketPC i Linux,
- telefony komórkowe z systemami operacyjnymi Symbian oraz Windows Mobile,
- konsole do gier i odtwarzacze iPod,
- tablety i smartfony działające pod kontrolą systemów Android firmy Google lub iOS firmy Apple,
- ... i wiele innych.

Podobnie do samego interpretera języka, również standardowe moduły biblioteki Pythona zaimplementowane są w taki sposób, by były maksymalnie przenośne. Co więcej, programy napisane w Pythonie są automatycznie kompilowane do pośredniego kodu bajtowego (ang. *byte code*), który działa tak samo na dowolnej platformie z zainstalowaną zgodną wersją Pythona (więcej szczegółowych informacji na ten temat znajdziesz w kolejnym rozdziale).

Oznacza to, że programy napisane w języku Python wykorzystujące podstawową składnię oraz standardowe biblioteki będą działały w ten sam sposób na systemach Linux, Windows i większości innych platform zawierających interpreter Pythona. Większość wersji Pythona posiada również rozszerzenia dla określonej platformy (na przykład obsługę COM w systemie Windows), jednak samo jądro Pythona wraz z bibliotekami standardowymi działają wszędzie tak samo. Jak wspomniano wcześniej, Python posiada również interfejs do zestawu narzędzi Tk GUI o nazwie `tkinter` (w Pythonie wersji 2.x — Tkinter), który pozwala na implementowanie w programach napisanych w tym języku pełnych, graficznych interfejsów użytkownika, działających na wszystkich najważniejszych platformach bez konieczności wprowadzania zmian do samego programu.

Ma duże możliwości

Z punktu widzenia możliwości Python jest swego rodzaju hybrydą. Jego zestaw narzędzi umieszcza go gdzieś pomiędzy tradycyjnymi językami skryptowymi (takimi, jak Tcl, Scheme oraz Perl) a językami służącymi do programowania systemowego (jak C, C++ oraz Java). Python ma w sobie całą prostotę i łatwość użycia języka skryptowego w połączeniu z bardziej zaawansowanymi narzędziami inżynierskimi, które zazwyczaj można znaleźć w językach kompilowanych. Z tego powodu Python — w odróżnieniu od niektórych języków skryptowych — świetnie nadaje się do realizacji dużych, rozbudowanych projektów programistycznych. Oto przegląd najważniejszych komponentów, które znajdziesz w zestawie narzędzi Pythona:

Typy dynamiczne

Python śledzi rodzaje obiektów wykorzystywane przez programy w czasie ich działania — nie wymaga zamieszczania skomplikowanych deklaracji typu czy rozmiaru w kodzie. Jak się przekonasz w rozdziale 6., w języku Python nie ma czegoś takiego, jak deklaracja typu czy zmiennej. Ponieważ kod napisany w Pythonie nie jest ograniczony do określonych typów danych, można go automatycznie stosować do całej gamy obiektów.

Automatyczne zarządzanie pamięcią

Python automatycznie alokuje obiekty i odzyskuje je (zwalnia pamięć), kiedy nie są już używane. Większość obiektów może rosnąć i kurczyć się na żądanie. Python automatycznie zarządza wykorzystaniem pamięci, tak aby użytkownik nie musiał tego robić sam.

Wspomaganie tworzenia dużych systemów

Python udostępnia wiele narzędzi, takich jak moduły, klasy oraz wyjątki, które znakomicie sprawdzają się przy budowaniu większych systemów. Narzędzia takie pozwalają na łączenie

fragmentów kodu w komponenty, wykorzystywanie programowania zorientowanego obiektowo, dostosowywanie kodu do własnych potrzeb oraz obsługę zdarzeń i błędów.

Wbudowane typy obiektów

Python udostępnia najczęściej wykorzystywane struktury danych, takie jak listy, słowniki oraz łańcuchy znaków — są one nieodłączną częścią samego języka. Jak się niebawem przekonasz, są one zarówno elastyczne, jak i łatwe w użyciu. Na przykład, wbudowane obiekty mogą rosnąć i zmniejszać się na żądanie, mogą być dowolnie zagnieżdżane w celu reprezentowania złożonych właściwości i nie tylko.

Wbudowane narzędzia

Do pracy z obiektami Python wyposażony jest w niezbędny zestaw standardowych operacji, takich jak konkatenacja (łączenie kolekcji), wyodrębnianie, sortowanie, odwzorowywanie i szereg innych.

Biblioteki narzędzi

Do wykonywania bardziej złożonych zadań Python dysponuje również sporym zestawem narzędzi, które obsługują niemal wszystko — od dopasowywania wyrażeń regularnych po zagadnienia sieciowe. Po opanowaniu podstaw programowania w Pythonie, duża część działań na poziomie aplikacji odbywa się z wykorzystaniem narzędzi z różnych bibliotek.

Narzędzia udostępniane przez niezależnych deweloperów

Ponieważ Python jest oprogramowaniem typu *open source*, niezależni programiści zachęcani są do udostępniania gotowych narzędzi, które wspomagają wykonywanie zadań wykraczających poza możliwości wbudowanych narzędzi tego języka. W Internecie można znaleźć darmowe biblioteki do obsługi technologii COM, przetwarzania obrazów, obliczeń numerycznych, XML, pracy z bazami danych i wykonywania wielu innych operacji.

Pomimo tak dużych możliwości oraz ogromnego zestawu dostępnych narzędzi Python nadal zachowuje swoją niezwykle prostą składnię i układ, dzięki czemu jest potężnym narzędziem programistycznym, zachowującym użyteczność i prostotę języków skryptowych.

Można go łączyć z innymi językami

Programy w Pythonie można z łatwością na różne sposoby łączyć z komponentami napisanymi w innych językach. Na przykład, API dla języka C w Pythonie pozwala na elastyczne wywoływanie programów napisanych w języku C z poziomu Pythona i odwrotnie. Oznacza to, że do systemów napisanych w Pythonie można łatwo dodawać nową funkcjonalność i wykorzystywać programy napisane w tym języku w innych środowiskach lub systemach.

Łączenie Pythona z bibliotekami napisanymi w językach takich, jak C czy C++ sprawia, że Python staje się łatwym w użyciu językiem, pozwalającym na dostosowywanie innych programów do własnych potrzeb. Jak wspomniano wcześniej, dzięki temu Python świetnie nadaje się do szybkiego tworzenia prototypów. Takie komponenty można najpierw zaimplementować w Pythonie, by skorzystać z szybkości programowania w tym języku, a następnie fragment po fragmencie przenieść je do języka C, spełniając odpowiednie wymagania w zakresie wydajności.

Jest względnie łatwy w użyciu

W porównaniu z językami takimi jak C++, Java czy C#, programowanie w Pythonie większości użytkowników wydaje się zadziwiająco proste. By uruchomić program napisany w Pythonie, wystarczy go napisać i wykonać. Nie ma tutaj żadnych przejściowych etapów komplikacji czy linkowania, które występują w językach takich, jak C czy C++. Python natychmiast wykonuje

programy, co sprawia, że programowanie w tym języku jest bardzo interaktywne, a *wyniki działania otrzymujemy bardzo szybko* — w wielu przypadkach efekty zmiany programu można zobaczyć niemal natychmiast po ich wprowadzeniu.

Oczywiście długość cyklu tworzenia oprogramowania to tylko jeden z aspektów łatwości użycia Pythona. Język ten celowo charakteryzuje się prostą składnią i posiada wbudowane narzędzia o dużych możliwościach. Niektórzy posuwają się nawet do określania Pythona mianem *wykonywalnego pseudokodu*. Ponieważ Python eliminuje większość złożoności spotykanych w innych językach programowania, programy napisane w tym Pythonie są prostsze, mniejjsze i bardziej elastyczne od swoich odpowiedników napisanych w innych popularnych językach.

Jest względnie łatwy do nauczenia się

Powysze stwierdzenie prowadzi nas bezpośrednio do kluczowego przesłania niniejszej książki — w porównaniu z innymi, powszechnie stosowanymi językami programowania, Python jest łatwy do opanowania. W praktyce, jeżeli jesteś doświadczonym programistą, możesz oczekwać, że będziesz pisał swoje pierwsze, niewielkie programy już po kilku dniach nauki, a podstawy Pythona opanujesz w ciągu zaledwie kilku godzin — nie powinieneś jednak oczekwać, że szybko staniesz się ekspertem tego języka (niezależnie od tego, co być może usłyszałeś w dziale marketingu...).

Oczywiście, opanowanie każdego zagadnienia tak obszernego jak dzisiejszy Python nie jest zadaniem trywialnym i dlatego poświęcimy na to całą pozostałą część tej książki. Warto jednak pamiętać, że naprawdę warto wiele zainwestować w opanowanie tego języka — w końcu pozwoli Ci to zdobyć umiejętności, które będą miały zastosowanie praktycznie w każdej dziedzinie programowania. Co więcej, większość użytkowników Pythona twierdzi, że krzywa uczenia się tego języka jest znacznie łagodniejsza niż w przypadku innych, popularnych języków programowania.

To dobra wiadomość dla zawodowych programistów chcących się nauczyć języka, który będą wykorzystywać w swojej pracy, a także dla użytkowników końcowych systemów, które udostępniają napisaną w Pythonie warstwę API, pozwalającą na dostosowania systemu do własnych celów czy zarządzanie nim. Obecnie wiele systemów wykorzystuje fakt, że użytkownicy mogą szybko opanować Pythona na wystarczającym poziomie, aby dostosowywać istniejące programy do własnych potrzeb z niewielkim wsparciem ze strony producenta (lub nawet przy jego braku). Co więcej, popularność Pythona spowodowała pojawienie się dużej grupy użytkowników programujących bardziej dla zabawy niż dla kariery, którym umiejętności rozwoju oprogramowania na pełną skalę po prostu nigdy nie będą potrzebne. Choć Python posiada szereg zaawansowanych narzędzi programistycznych, opanowanie podstaw tego języka jest jednakowo proste zarówno dla osób początkujących, jak i doświadczonych deweloperów.

Zawdzięcza swoją nazwę Monty Pythonowi

Jasne, trudno to nazwać techniczną mocną stroną, ale w świecie Pythona kwestią ta wydaje się być zaskakująco dobrze ukrywanym sekretem, który właśnie teraz chciałbym ujawnić. Pomimo tych wszystkich gadów, płazów i ssaków często prezentowanych na okładkach książek dotyczących Pythona, prawda jest taka, że swoją nazwę język ten zawdzięcza słynnej brytyjskiej grupie komediowej Monty Python — twórcom wspaniałego serialu komediowego BBC z lat 70. ubiegłego wieku, zatytułowanego *Latający Cyrk Monty Pythona* (*Monty Python's Flying Circus*) oraz kilku późniejszych filmów pełnometrażowych, takich jak *Monty Python i Święty Graal*, który do dziś bardzo popularny. Twórca języka Python, Guido van Rossum, podobnie jak wielu innych programistów i deweloperów, jest wielkim fanem Monty Pythona (rzeczywiście, wydaje się, że poczucie humoru prezentowane przez grupę Monty Pythona wydaje się być wszystkim twórcom oprogramowania szczególnie bliskie...).

Takie dziedzictwo bez wątpienia dodaje nutkę brytyjskiego humoru do przykładów pisanych w języku Python. Tradycyjne `foo` i `bar`, zwyczajowo spełniające rolę przykładowych nazw zmiennych, w Pythonie są na przykład zastępowane przez `spam` (z ang. *mielonka*) i `eggs` (z ang. *jajka*). Okazjonalnie pojawiają się również inne perełki, takie jak `Brian`, ni czy `shrubbery` (fani Monty Pythona doskonale będą wiedzieć, o które filmy chodzi). Dziedzictwo tej grupy komików wywarło także ogromny wpływ na całą społeczność użytkowników Pythona — przykładowo niektóre wydarzenia czy odczyty na konferencjach poświęconych temu językowi często noszą nazwy takie jak *The Spanish Inquisition* (z ang. hiszpańska inkwizycja).

Oczywiście wszystko to jest zabawne tylko wtedy, gdy wiesz, o co chodzi. Do zrozumienia przykładów czerpiących z Monty Pythona (w tym tych prezentowanych w niniejszej książce) nie jest jednak potrzebna znajomość tego serialu, ale dzięki powyższym objaśnieniom przynajmniej będziesz wiedział, skąd to się wszystko wzięło (hej, pamiętaj, że Cię ostrzegałem — poczucie humoru Monty Pythona jest bardzo zaraźliwe...).

Jak Python wygląda na tle innych języków?

Aby umieścić Pythona w jakimś znanym sobie kontekście, wielu użytkowników często porównuje go z innymi językami, takimi jak Perl, Tcl czy Java. W tej sekcji spróbujemy dokonać pewnego podsumowania takich porównań.

Chciałbym z góry zaznaczyć, że nie jestem fanem wygrywania poprzez dyskredytowanie konkurencji — nie działa to na dłuższą metę i takie podejście nie jest też naszym celem. Co więcej, to nie będzie gra o sumie zerowej — większość programistów w swojej pracy będzie korzystać z wielu języków programowania. Nie zmienia to jednak w niczym faktu, że narzędzia programistyczne przedstawiają różne wybory i kompromisy, które zasługują na uwagę. W końcu, jeżeli Python nie miałby do zaoferowania czegoś więcej niż konkurencja, nigdy nie zostałby użyty jako narzędzie pierwszego wyboru.

O kompromisach wydajności Pythona pisaliśmy już wcześniej, więc tutaj skupimy się na jego funkcjonalności. Choć istnieje wiele różnych, znakomitych języków programowania, które warto znać i używać, wiele osób uważa, że:

- Python ma większe możliwości od języka *Tcl*. Silne wsparcie Pythona dla tworzenia dużych projektów programistycznych sprawia, że ma on zastosowanie w rozwoju większych systemów, a jego biblioteka narzędzi aplikacyjnych jest bardzo rozbudowana.
- Kod języka Python jest bardziej czytelny niż *Perl*. Python ma jasną składnię i prosty, spójny projekt, co powoduje, że jego kod może być wykorzystywany wielokrotnie, jest łatwiejszy w utrzymaniu i modyfikacji, a także pomaga zmniejszyć liczbę błędów programu.
- Python jest prostszy i łatwiejszy w użyciu niż *Java* i *C#*. Python jest językiem skryptowym, a Java i C# dziedziczą wiele złożonych elementów składni z dużych, zorientowanych obiektowo języków systemowych, takich jak *C++*.
- Python jest prostszy i łatwiejszy w użyciu od *C++*. Kod Pythona jest prostszy niż jego odpowiednik w *C++* i często o dwadzieścia do nawet trzydziestu procent mniejszy, choć jako język skryptowy Python czasami spełnia różne role.
- Python jest prostszy i jest językiem wyższego poziomu niż *C*. Odłączenie Pythona od architektury sprzętowej sprawia, że jego kod jest mniej skomplikowany, lepiej zorganizowany i bardziej przystępny niż kod języka C, będącego prekursorem *C++*.
- Python jest bardziej wydajny, uniwersalny i wieloplatformowy niż *Visual Basic*. Jest bogatszym i znacznie szerzej stosowanym językiem, a jego natura *open source* oznacza, że nie jest kontrolowany przez jedną firmę.
- Python jest bardziej czytelny i bardziej ogólnego przeznaczenia niż *PHP*. Choć Python jest również wykorzystywany do tworzenia stron internetowych, ale oprócz tego jest

powszechnie stosowany jest w niemal każdej innej dziedzinie informatyki, od robotyki po animację filmów i gry komputerowe.

- Python jest bardziej wydajny i uniwersalny niż *JavaScript*. Ma większy zestaw narzędzi i nie jest tak ściśle powiązany z tworzeniem stron internetowych. Jest również używany do modelowania naukowego, obsługi oprzyrządowania i w wielu innych zastosowaniach.
- Python jest bardziej dojrzały i ma bardziej czytelną składnię od języka *Ruby*. Składnia Pythona jest bardziej przejrzysta, zwłaszcza w złożonym kodzie, a jego zorientowanie obiektowe jest w pełni opcjonalne dla użytkowników i projektów, w których nie znajduje ono zastosowania.
- Python jest bardziej dojrzały i wszechstronny niż *Lua*. Rozbudowany zestaw funkcji Pythona i ogromna liczba różnych bibliotek dają mu znacznie szerszy obszar zastosowań niż *Lua* czy innym językiem skryptowym, takim jak *Tcl*.
- Python jest mniej ezoteryczny niż *Smalltalk*, *Lisp* czy *Prolog*. Posiada dynamiczny charakter języków takich, jak *SmallTalk* czy *Lisp*, ale łączy go z prostą, tradycyjną składnią, dostępną zarówno dla doświadczonych programistów jak i zwykłych użytkowników, chcących dostosować wybrane systemy do własnych potrzeb.

Wiele osób uważa, że Python sprawdza się lepiej od innych, dostępnych dzisiaj języków programowania, w szczególności w przypadku programów, które robią coś więcej, niż tylko przeszukują pliki tekstowe, i które w przyszłości mogą być analizowane i modyfikowane przez inne osoby (lub przez nas samych!). Co więcej, jeśli nasza aplikacja nie wymaga jakiejś wyjątkowej wydajności, Python jest często rozsądnią alternatywą dla języków systemowych, takich jak C, C++ czy Java — programy napisane w Pythonie będzie bowiem łatwiej napisać, utrzymywać i usuwać z nich błędy.

Oczywiście autor niniejszej książki jest znanym promotorem Pythona od 1992 roku, dlatego powyższe stwierdzenia możesz potraktować dość swobodnie, a opinie miłośników innych języków programowania mogą od nich znaczco odbiegać — nie zmienia to jednak w niczym faktu, że odzwierciedlają one jednak odczucia współdzielone przez wielu programistów, którzy poświęcili swój czas na poznanie tego, co może dać im Python.

Podsumowanie rozdziału

Na tym kończymy marketingową część niniejszej książki. W tym rozdziale omówiliśmy niektóre powody, dla których ludzie często wybierają Pythona do realizacji swoich zadań programistycznych. Dowiedzieliśmy się również, jaki jest zakres zastosowań tego języka i zapoznaliśmy się z reprezentatywną próbką jego użytkowników. Moim założeniem jest jednak nauczanie Pythona, a nie sprzedawanie go. Najlepszym sposobem oceny języka jest zobaczenie go w działaniu, dlatego pozostała część książki skupia się w całości na elementach języka, o których tutaj jedynie wspomnialiśmy.

Kolejne dwa rozdziały rozpoczynają techniczne wprowadzenie do języka Python. Dowieemy się z nich, jak uruchamia się programy napisane w tym języku, przyjrzymy się modelowi wykonywania kodu bajtowego Pythona, a także omówimy podstawowe zagadnienia związane z plikami modułów, w których zapisywany jest kod. Nadrzędnym celem będzie dostarczenie Ci wystarczającej ilości informacji, które pozwolą na uruchamianie przykładów i ćwiczeń z dalszej części książki. Prawdziwe programowanie zaczniemy dopiero w rozdziale 4. — najpierw musisz opanować podstawy języka Python.

Sprawdź swoją wiedzę — quiz

W tym wydaniu książki każdy rozdział kończy się krótkim quizem dotyczącym zaprezentowanego materiału, który pomoże Ci w powtórzeniu i zapamiętaniu omawianych zagadnień. Odpowiedzi na pytania znajdują się bezpośrednio pod nimi, ale zachęcam do zapoznania się z nimi dopiero po samodzielny udzieleniu odpowiedzi na poszczególne pytania.

Poza quizami kończącymi rozdziały na końcu każdej części książki znajdują się *ćwiczenia* zaprojektowane tak, abyś mógł zacząć samodzielne programowanie w Pythonie. A oto pierwszy test — powodzenia i pamiętaj, że w razie potrzeby zawsze możesz wrócić do materiału omawianego w danym rozdziale.

1. Podaj sześć głównych powodów, dla których ludzie wybierają Pythona.
2. Wymień cztery znaczące firmy czy organizacje, które wykorzystują Pythona.
3. Dlaczego można *nie chcieć* używać Pythona w aplikacji?
4. Do czego można wykorzystać Pythona?
5. Do czego służy polecenie `import this` w języku Python?
6. Dlaczego słowo `spam` (z ang. „mielonka”) pojawia się w tak wielu przykładach kodu napisanego w języku Python, publikowanych w książkach oraz w Internecie?
7. Jaki jest Twój ulubiony kolor?

Sprawdź swoją wiedzę — odpowiedzi

I jak poszło? Poniżej znajdują się odpowiedzi, których udzieliłbym ja sam, choć w przypadku niektórych pytań z quizów poprawnych odpowiedzi może być więcej. Ponownie zachęcam do zapoznania się z odpowiedziami — nawet jeśli jesteś pewny, że odpowiedziałeś poprawnie. W rozwiązańach często znajdziesz dodatkowe informacje. Jeżeli któraś z odpowiedzi jest niezrozumiała, zawsze możesz wrócić do odpowiedniego fragmentu tekstu rozdziału.

1. Jakość oprogramowania, wydajność programistów, przenośność programu, dodatkowe biblioteki, integracja komponentów oraz po prostu zadowolenie. Z tych wszystkich opcji najczęstszymi powodami, dla których użytkownicy wybierają Pythona, są jakość oraz wydajność.
2. Google, Industrial Light & Magic, CCP Games, Jet Propulsion Labs, Maya, ESRI i wiele innych. Praktycznie wszystkie firmy i organizacje zajmująca się programowaniem wykorzystują w jakimś stopniu Pythona — albo w strategicznym, długofalowym rozwijaniu produktów, albo w zadaniach krótkoterminowych, takich jak testowanie czy administrowanie systemami.
3. Najpoważniejszą słabą stroną Pythona jest jego wydajność — zazwyczaj nie będzie działał tak szybko, jak języki w pełni komplikowane, takie jak C czy C++. Z drugiej strony, Python jest wystarczająco szybki dla wielu zastosowań, a typowy kod w Pythonie działa z prędkością zbliżoną do języka C, ponieważ interpreter Pythona wykorzystuje skompilowany kod napisany w C. Jeżeli jednak szybkość jest kwestią kluczową, dla najbardziej wymagających części aplikacji dostępne są skompilowane rozszerzenia.
4. Pythona można używać prawie do wszystkiego, co wykonuje się za pomocą komputera — od tworzenia stron internetowych i gier po robotykę i sterowanie pojazdami kosmicznymi.

5. O tym wspominaliśmy w przypisie: instrukcja `import this` powoduje w Pythonie wyświetlenie kilku założeń filozofii będącej fundamentem projektu tego języka. W kolejnym rozdziale pokażemy, jak można wykonać tę instrukcję.
6. „Spam” („mielonka”) to odniesienie do sławnego skeczu *Latającego Cyrku Monty Pythona*, w którym ludzie próbujący zamówić w restauracji coś do jedzenia są zagłuszani przez chór Wikingów śpiewających o mielonce. Przy okazji `spam` to również popularna nazwa zmiennej stosowanej w wielu przykładowych skryptach napisanych w języku Python.
7. Niebieski. Nie, żółty! (Zobacz poprzednią odpowiedź...).

Python to inżynieria, nie sztuka

Kiedy Python pojawił się po raz pierwszy, we wczesnych latach dziewięćdziesiątych, wraz z nim pojawił się również klasyczny już konflikt między jego zwolennikami a fanami innego popularnego języka skryptowego — Perla. Osobiście uważam, że cała ta debata jest już nudna i nie ma żadnego uzasadnienia — programiści są przecież na tyle intelligentni, aby samodzielnie wyciągnąć własne wnioski. Ponieważ jednak w czasie szkoleń jest to jedna z najczęściej poruszanych kwestii, napisanie kilku słów na ten temat wydaje się konieczne.

Krótko mówiąc: *w Pythonie można zrobić wszystko to, co w Perlu, z tym, że po zakończeniu da się przeczytać i zrozumieć gotowy kod.* I tyle. Dziedziny zastosowania tych języków w dużej mierze się pokrywają, jednak Python jest bardziej zorientowany na tworzenie czytelnego kodu. Dla wielu osób zwiększała czytelność kodu Pythona przekłada się na większe możliwości ponownego użycia i łatwiejszą konserwację kodu, dzięki czemu Python jest lepszym wyborem dla programów, które nie zostaną skasowane po jednorazowym użyciu. Kod w Perlu łatwo jest napisać, jednak trudniej go analizować i modyfikować. Pamiętając o tym, że większość programów ma dużo dłuższy żywot, niż to na początku planujemy, wiele osób uważa Pythona za wydajniejsze narzędzie.

By nieco bardziej rozwinać powyższe stwierdzenia, warto dodać, że oba języki odzwierciedlają doświadczenia i wykształcenie swoich projektantów, co pokazuje również, jakie są przyczyny wybierania Pythona przez niektóre osoby. Twórca Pythona jest matematykiem, który stworzył język z wysokim stopniem jednorodności — jego składnia i zestaw narzędzi są bardzo spójne. Tak jak matematyka — projekt tego języka jest ortogonalny, większa część języka pochodzi od niewielkiego zbioru podstawowych koncepcji. Przykładowo, kiedy zrozumiesz polimorfizm w Pythonie, cała reszta to tylko szczegóły. Twórca Perla jest za to jazykoznawcą, a sam język znakomicie to odzwierciedla. W Perlu te same zadania można wykonać na wiele sposobów, konstrukcje tego języka wchodzą ze sobą w interakcje w sposób kontekstowy i czasami dość subtelny — podobnie do języków naturalnych. Dobrze znane motto Perla mówi: „Istnieje więcej niż jeden sposób wykonania danego zadania”. Dzięki takiemu projektowi zarówno sam język, jak i społeczność jego użytkowników od zawsze zachęcali do wolności wyrażania się w czasie tworzenia kodu. Kod napisany w Perlu przez jedną osobę może być radykalnie inny od kodu napisanego przez kogoś innego. Tak naprawdę pisanie unikalnego, celowo zagmatwanego kodu jest często wśród użytkowników Perla powodem do dumy.

Jak przyzna każdy, kto miał do czynienia z koniecznością analizowania i modyfikowania większych ilości kodu, *wolność wyrażania się jest świetna dla artysty, ale jednak niekoniecznie dla programisty czy inżyniera.* W inżynierii potrzebny jest minimalny zestaw narzędzi i przewidywalność. W programowaniu wolność wyrażania się może prowadzić do koszmaru przy aktualizacji kodu. Jak zdradził mi niejeden użytkownik Perla, rezultatem zbyt dużej wolności jest często to, że kod o wiele łatwiej jest napisać od nowa, niż próbować go modyfikować. Takie podejście jest jednak dalekie od doskonałości.

Pomyślmy o tym w taki sposób: kiedy ludzie tworzą obraz czy rzeźbę, robią to dla siebie samych, dla czysto estetycznych celów. Możliwość, że ktoś inny będzie kiedyś musiał zmieniać nasze dzieło, mało komu przychodzi do głowy. To właśnie kluczowa różnica

między sztuką a inżynierią. Kiedy ludzie piszą oprogramowanie, nie robią tego dla siebie samych. Tak naprawdę nie robią tego nawet dla komputera. Dobrzy programiści wiedzą, że kod pisany jest dla kolejnych osób, które będą go musiały odczytywać, by z niego korzystać i go utrzymywać. Jeśli taka osoba nie będzie w stanie zrozumieć kodu, w rzeczywistych warunkach stanie się on bezużyteczny. Innymi słowy, programowanie nie polega na tworzeniu złożonego, zagmatwanego kodu — *chodzi o to, aby kod w jasny sposób komunikował swoje przeznaczenie*.

W tym właśnie punkcie wiele osób zauważa, jak bardzo Python różni się od języków skryptowych, takich jak Perl. Ponieważ model składni Pythona zmusza użytkownika do pisania czytelnego kodu, programy napisane w tym języku o wiele lepiej nadają się do pełnego cyklu tworzenia oprogramowania. A ponieważ Python podkreśla takie koncepcje, jak ograniczona interakcja, jednorodność, regularność i spójność, o wiele bardziej bezpośrednio wymusza tworzenie kodu, który może być używany przez długi czas po napisaniu.

Na dłuższą metę sam nacisk na *jakość kodu* w Pythonie zwiększa wydajność programistów, a także ich zadowolenie. Oczywiście programiści tworzący kod w tym języku mogą być bardzo kreatywni, a jak zobaczymy niebawem, Python również może oferować wiele rozwiązań dla niektórych zadań — czasami nawet więcej niż powinien dzisiaj, z czym będziemy musieli się zmierzyć również w tej książce. W praktyce informacje zamieszczone w tej ramce mogą być traktowane jako swego rodzaju *ostrzeżenie*, że jakość oprogramowania jest tworem bardzo wrażliwym, mocno uzależnionym zarówno od technologii, jak i od samych ludzi. W przeciwnieństwie do wielu innych języków, Python od samego początku w samej swojej filozofii kładł duży nacisk na tworzenie dobrego oprogramowania, ale zawsze powinieneś pamiętać, że cała reszta zależy od Ciebie.

Z argumentami przedstawionymi powyżej zgadza się wielu ludzi, którzy wybrali Pythona. Każdy użytkownik powinien jednak wyciągnąć własne wnioski na podstawie tego, co może im zaoferować ten język. Aby zatem rozpocząć swoją przygodę z Pythonem, przejdź do kolejnego rozdziału.

[1] Aby lepiej poznać filozofię tego języka, w wierszu poleceń konsoli Pythona wpisz polecenie `import this` (w rozdziale 3. dowiesz się, jak to zrobić). Polecenie to wywołuje ukrytą w interpreterze niespodziankę — zbiór zasad projektowania, które w bardzo obrazowy sposób opisują piękno, prostotę i przejrzystość tego wspariałego języka programowania i są wyznacznikiem postępowania dla całej społeczności jego „wyznawców”. Często spotykany akronim EIBTI jest modnym żargonem oznaczającym „jawne jest lepsze od niejawnego” (ang. *explicit is better than implicit*). Choć oczywiście wspomniane zasady nie są żadnym „dekalgiem”, ale w praktyce są wystarczająco powszechnie uznawane, aby zakwalifikować je jako motto i credo języka Python, które będziemy często cytować w tej książce.

Rozdział 2. Jak Python wykonuje programy?

Niniejszy rozdział wraz z kolejnym omawiają wykonywanie programów — sposób uruchamiania kodu oraz wykonywania go przez Pythona. W tym rozdziale dowiesz się, jak ogólnie przebiega uruchamianie programów w interpreterze Pythona, a w rozdziale 3. nauczysz się uruchamiać własne programy.

Szczegóły przebiegu procesu uruchamiania programu są z natury specyficzne dla danej platformy, więc niektóre materiały z tych dwóch rozdziałów mogą nie mieć zastosowania do systemu, w jakim pracujesz, dlatego bardziej zaawansowani czytelnicy mogą spokojnie ominąć fragmenty, które ich nie dotyczą. Podobnie osoby, które w przeszłości korzystały już z podobnych narzędzi i wolą od razu przejść do opisu samego języka, mogą z zupełnie czystym sumieniem pominąć ten rozdział, w którym już za chwilę dowiesz się, w jaki sposób Python uruchamia kod programu, i to zanim jeszcze nauczysz się takie programy pisać.

Wprowadzenie do interpretera Pythona

Dotychczas mówiliśmy o Pythonie jako o języku programowania. W obecnej implementacji Python jest także pakietem oprogramowania zwanym *interpreterem*. Interpreter to rodzaj programu, który wykonuje inne programy. Kiedy pisze się kod w Pythonie, interpreter tego języka odczytuje następnie program i wykonuje zawarte w nim instrukcje. W rezultacie interpreter jest warstwą logiki oprogramowania znajdująca się pomiędzy kodem a urządzeniami naszego komputera.

Po zainstalowaniu pakietu języka Python na komputerze generowanych jest wiele komponentów — minimalny zestaw składa się z interpretera i biblioteki pomocniczej. W zależności od sposobu wykorzystania interpreter Pythona może przybrać postać programu wykonywalnego lub zbioru bibliotek połączonych z innym programem. W zależności od wersji Pythona interpreter może być programem w języku C, zbiorem klas Javy czy jeszcze czymś innym. Bez względu na formę kod w Pythonie musi zawsze być wykonywany przez interpreter. Aby to było możliwe, musisz zainstalować interpreter Pythona na swoim komputerze.

Szczegóły instalacji Pythona różnią się w zależności od platformy i są omówione dokładniej w dodatku A. W skrócie wygląda to tak:

- Użytkownicy systemu Windows pobierają i uruchamiają wykonywalny plik instalacyjny umieszczający Pythona na ich komputerach. Wystarczy dwukrotnie kliknąć plik instalatora, a później na wszystkie pytania odpowiadać *Yes (Tak)* lub *Next (Dalej)*.
- Użytkownicy systemów Linux oraz macOS naprawdopodobnie mają już Pythona zainstalowanego na swoich komputerach — jest on obecnie standardowym komponentem tych platform.
- Niektórzy użytkownicy Linuksa i macOS (oraz większość użytkowników Uniksa) samodzielnie komplilują Pythona z pakietu dystrybucyjnego zawierającego pełny kod źródłowy.
- Użytkownicy Linuksa mogą również skorzystać z plików RPM, natomiast dla użytkowników systemu macOS dostępne są pakiety instalacyjne specyficzne dla Maca.

- Dla innych platform istnieją techniki instalacyjne odnoszące się tylko do nich. Python jest na przykład dostępny na telefony komórkowe, tablety, konsole do gier i iPody, ale szczegóły poszczególnych procedur instalacji mogą być bardzo zróżnicowane.

Samego Pythona można pobrać z odpowiedniej podstrony witryny internetowej tego języka — <http://www.python.org>. Można go również uzyskać za pomocą różnych kanałów dystrybucji. Należy pamiętać, aby przed instalacją zawsze sprawdzić, czy pakiet ten nie jest już zainstalowany na naszym komputerze. Jeżeli pracujesz w systemie Windows 7 lub starszym, zrobisz to tak, jak pokazano na rysunku 2.1; poszczególne opcje menu omówimy w kolejnym rozdziale. W systemach Unix oraz Linux zazwyczaj można znaleźć Pythona w drzewie katalogu `/usr`.

Rysunek 2.1. W systemie Windows 7 i starszych, Python widoczny jest w menu Start. W różnych wydaniach Pythona może to wyglądać nieco inaczej, jednak zazwyczaj IDLE służy do uruchamiania graficznego środowiska programistycznego, a opcja Python powoduje rozpoczęcie prostej sesji interaktywnej. W menu tym znajduje się również standardowa dokumentacja (Python Manuals), a także silnik dokumentacji PyDoc (Module Docs). Więcej informacji na temat Pythona w systemach Windows 8 i innych znajdziesz w rozdziale 3. i dodatku A

Ponieważ szczegóły instalacji na poszczególnych platformach mogą się od siebie nieco różnić, nie będziemy się nad tym dalej tutaj rozwodzić. Więcej informacji na temat procesu instalacji znajdziesz w dodatku A. Na potrzeby tego i kolejnego rozdziału przyjmiemy po prostu założenie, że masz już zainstalowanego Pythona na swoim komputerze.

Wykonywanie programu

To, co oznacza napisanie i wykonanie skryptu w Pythonie, zależy od tego, czy patrzymy na te zadania z punktu widzenia programisty, czy z punktu widzenia interpretera Pythona. Oba punkty widzenia oferują interesującą perspektywę programowania w Pythonie.

Z punktu widzenia programisty

W najprostszej formie program w Pythonie jest zwykłym plikiem tekstowym zawierającym instrukcje w tym języku. Poniższy plik o nazwie `script0.py` jest jednym z prostszych skryptów, jakie moglibyśmy sobie wyobrazić, a mimo to jest pełnoprawnym programem w języku Python:

```
print('Witaj, świecie!')  
print(2 ** 100)
```

Plik ten składa się z dwóch instrukcji `print` języka Python, które służą do wyświetlenia (wydrukowania) łańcucha znaków (tekstu znajdującego się w apostrofach) oraz wyniku wyrażenia liczbowego (2 do potęgi 100) do strumienia wyjścia. Nie przejmuj się teraz składnią poleceń — w tym rozdziale interesuje nas jedynie sposób wykonywania kodu. W dalszej części książki omówiona zostanie instrukcja `print` i wyjaśnione to, dlaczego można podnieść 2 do potęgi 100 bez przepełnienia.

Plik zawierający taki zestaw poleceń można utworzyć w dowolnym edytorze tekstu. Zgodnie z konwencją pliki Pythona otrzymują nazwy kończące się rozszerzeniem `.py`. Z technicznego punktu widzenia taki schemat nazewnictwa jest wymagany jedynie dla plików, które będą importowane, co zostanie omówione w dalszej części książki, jednak w praktyce przyjęło się, że pliki zawierające kod Pythona posiadają rozszerzenie `.py`.

Po wpisaniu tych instrukcji do pliku tekstowego trzeba przekazać Pythonowi, że ma ten plik *uruchomić* — co oznacza wykonanie wszystkich instrukcji z pliku od góry do dołu, jedna po drugiej. Jak zobaczymy w kolejnym rozdziale, pliki programów w języku Python można wykonywać z poziomu wiersza poleceń powłoki, klikając ich ikony, uruchamiać je w zintegrowanym środowisku programistycznym (IDE), a także za pomocą innych, standardowych technik. Jeżeli wszystko pójdzie dobrze, po uruchomieniu programu gdzieś na ekranie zobaczymy wynik działania dwóch poleceń `print` — domyślnie będzie to zazwyczaj w tym samym oknie, z którego uruchamialiśmy program:

```
Witaj, świecie!
```

```
1267650600228229401496703205376
```

A oto, co stało się, kiedy wykonalem ten skrypt z wiersza poleceń na laptopie działającym pod kontrolą systemu Windows, aby upewnić się, że nie popełniłem żadnych głupich literówek:

```
C:\temp> python script0.py
```

```
Witaj, świecie!
```

```
1267650600228229401496703205376
```

Pełny opis tego procesu znajdziesz w rozdziale 3. Zamieściliśmy tam wiele ciekawych informacji dotyczących pisania i uruchamiania programów; może to być dla Ciebie interesujące zwłaszcza jeżeli dopiero zaczynasz przygodę z programowaniem. Na chwilę obecną po prostu przyjmujemy, że udało nam się teraz wykonać skrypt wyświetlający ciąg znaków oraz wartość liczbową. Najprawdopodobniej taki kod nie zapewniłby nam zwycięstwa w żadnym konkursie programistycznym, ale w zupełności wystarczy, aby zrozumieć podstawy wykonywania programów.

Z punktu widzenia Pythona

Krótki opis z poprzedniego podrozdziału jest dosyć typowy dla języków skryptowych i generalnie jest to najczęściej wszystko, co muszą wiedzieć programiści Pythona. Kod programu umieszczamy w plikach tekstowych, które wykonujemy za pośrednictwem interpretera. Kiedy jednak dajemy Pythonowi sygnał do działania, „pod maską” interpretera dzieje się trochę więcej. Choć znajomość wewnętrznych mechanizmów Pythona nie jest ściśle wymagana do programowania w tym języku, ich zrozumienie Pythona może pomóc nam zobaczyć wykonywanie programów w szerszej perspektywie.

Kiedy nakazujemy Pythonowi uruchomić skrypt, przed rozpoczęciem wykonywania kodu Python przeprowadza kilka operacji. Kod źródłowy jest najpierw kompilowany na tak zwany kod bajtowy (ang. *byte code*), a następnie przesyłany do czegoś o nazwie „maszyna wirtualna”.

Kompilacja kodu bajtowego

Kiedy wykonujemy program, Python w sposób prawie całkowicie przed nami ukryty najpierw kompiluje *kod źródłowy* (instrukcje znajdujące się w pliku) do formatu znanego jako *kod bajtowy*. Kompilacja to po prostu tłumaczenie kodu na inny format, a kod bajtowy jest niskopoziomową, niezależną od platformy reprezentacją kodu źródłowego. Python przekłada każdą z instrukcji źródłowych na grupę instrukcji kodu bajtowego poprzez podzielenie ich na pojedyncze kroki. Proces przekładania kodu źródłowego na kod bajtowy odbywa się z myślą o szybkości wykonania — kod bajtowy może działać o wiele szybciej od oryginalnych instrukcji z kodu źródłowego zawartego w pliku tekstowym.

W poprzednim akapicie wspomniałem, że proces ten jest *prawie* całkowicie przed nami ukryty. Jeżeli proces Pythona ma uprawnienia do zapisu na naszym komputerze, kod bajtowy programu zostanie zapisany w pliku z rozszerzeniem `.pyc` (`.pyc` oznacza skompilowane źródło `.py`). W Pythonie w wersjach niższych niż 3.2, pliki takie pojawiają się po uruchomieniu programu obok

odpowiadających im plików źródłowych (w tych samych katalogach). Na przykład po uruchomieniu pliku `script.py` w tym samym katalogu pojawi się plik `script.pyc`.

W wersji 3.2 i nowszych Python zapisuje pliki kodu bajtowego `.pyc` w podkatalogu o nazwie `_pycache_` (znajdującym się w katalogu z plikami źródłowymi), nadając im nazwy identyfikujące wersję Pythona, przy użyciu której zostały utworzone (np. `script.cpython-33.pyc`). Nowy podkatalog `_pycache_` pomaga uniknąć bałaganu, a nowa konwencja nazewnictwa plików kodu bajtowego uniemożliwia różnym wersjom Pythona zainstalowanym na tym samym komputerze wzajemne nadpisywania zapisanego kodu bajtowego. Modele plików kodu bajtowego przeanalizujemy bardziej szczegółowo w rozdziale 22., choć Python tworzy je automatycznie i nie są one aż tak istotne w przypadku większości pisanych w nim programów; ponadto mogą w nich występuwać znaczące różnice między różnymi wersjami Pythona.

W obu przypadkach Python zapisuje taki kod bajtowy w celu optymalizacji szybkości wykonania. Następnym razem, kiedy będziemy wykonywać dany program, Python załaduje pliki `.pyc` i pominie etap komplikacji — o ile oczywiście nie zmieniliśmy kodu źródłowego od czasu, gdy kod bajtowy był zapisany po raz ostatni i nie próbujemy uruchomić tego programu przy użyciu innej wersji Pythona niż ta, która została użyta do utworzenia kodu bajtowego. Działa to w następujący sposób:

- Zmiany w kodzie źródłowym — Python automatycznie sprawdza znaczniki czasu ostatniej modyfikacji plików kodu źródłowego i bajtowego, aby wiedzieć, kiedy należy je ponownie skompilować — jeżeli wprowadzisz zmiany w kodzie źródłowym i zapiszesz je w pliku, kod bajtowy zostanie automatycznie ponownie utworzony przy następnym uruchomieniu programu.
- Wersje Pythona — Podczas importowania kodu interpreter sprawdza również, czy kod bajtowy musi zostać ponownie skompilowany, ponieważ został utworzony przez inną wersję Pythona. Interpreter używa do tego celu „magicznego” numeru wersji, zapisanego w samym pliku kodu bajtowego (wersja 3.2 i wcześniejsze), albo informacji o wersji zawartych w nazwach plików kodu bajtowego (wersje nowsze niż 3.2).

W rezultacie zarówno zmiany w kodzie źródłowym, jak i użycie innej wersji Pythona automatycznie powodują utworzenie nowego pliku kodu bajtowego. Jeżeli z jakiegoś powodu Python nie może zapisać kodu bajtowego na dysku Twojego komputera, sam program nadal będzie działał poprawnie — kod bajtowy zostanie utworzony w pamięci i po prostu usunięty po zakończeniu działania programu. Ponieważ jednak pliki `.pyc` przyspieszają rozpoczęcie wykonywania programu, w przypadku większych aplikacji warto upewnić się, że są one zapisywane. Pliki kodu bajtowego to także jeden ze sposobów udostępniania programów napisanych w tym języku — Python z powodzeniem uruchomi program składający się z samych plików `.pyc`, nawet kiedy oryginalne pliki `.py` nie są dostępne. Więcej informacji na temat innych opcji dystrybucji programów napisanych w języku Python znajdziesz w podrozdziale „Zamrożone pliki binarne”.

Trzeba także pamiętać, że kod bajtowy jest zapisywany w plikach tylko dla *importowanych* plików kodu źródłowego, a nie dla głównych plików programu, uruchamianych tylko jako skrypty (ściśle mówiąc, jest to optymalizacja importu). Podstawowe zagadnienia związane z importowaniem kodu źródłowego omówimy w rozdziale 3., a jeszcze dokładniej przyjrzymy się im w części V tej książki. Warto pamiętać, że dany plik jest importowany (i ewentualnie komplikowany) *tylko raz* na uruchomienie programu, a kod bajtowy nigdy nie jest zapisywany dla kodu wpisywanego z poziomu *interaktywnej sesji* Pythona — jest to tryb programowania, o którym więcej opowiemy w rozdziale 3.

Maszyna wirtualna Pythona

Po skompilowaniu programu do postaci kodu bajtowego (lub załadowaniu kodu bajtowego z istniejących plików `.pyc`), jest on przesyłany do wykonania do czegoś, co znane jest pod nazwą *maszyny wirtualnej Pythona* (PVM — ang. *Python Virtual Machine*). Maszyna wirtualna Pythona brzmi bardzo poważnie, tak naprawdę jednak nie jest to ani oddzielny program, ani coś, co musi być osobno instalowane. PVM to generalnie wielka pętla, która przechodzi przez kolejne

instrukcje kodu bajtowego, jedna po drugiej i wykonuje działania i operacje z nimi związane. Maszyna wirtualna Pythona jest silnikiem wykonawczym (ang. *runtime engine*) tego języka. Jest zawsze obecna jako część systemu Pythona i jest komponentem, który odpowiedzialny jest za samo wykonywanie skryptów. Z technicznego punktu widzenia jest ostatnim etapem działania tego, co nazywamy interpreterem Pythona.

Na rysunku 2.2 widać opisaną powyżej strukturę wykonawczą Pythona. Należy pamiętać, że ten cały, złożony komponent jest zupełnie niewidoczny dla programistów. Kompilacja kodu bajtowego odbywa się automatycznie, a PVM jest częścią systemu Pythona, który instaluje się na komputerze. Programiści po prostu piszą kod i uruchamiają swoje programy, a interpreter Pythona sam zajmuje się całą resztą.

Rysunek 2.2. Tradycyjny model wykonywania kodu Pythona: pisany przez programistę kod źródłowy jest przekładany na kod bajtowy, który następnie jest wykonywany za pomocą maszyny wirtualnej Pythona. Kod jest komplikowany automatycznie, a następnie interpretowany

Wpływ na wydajność

Czytelnicy znający języki w pełni komplikowane, takie jak C czy C++, mogą zauważać kilka cech odróżniających je od modelu zastosowanego w Pythonie. Przykładowo, w przypadku programów napisanych w języku Python nie mamy do czynienia z etapem budowania kodu (ang. *build, make*) — kod uruchamiany jest natychmiast po napisaniu. Kod bajtowy Pythona nie jest również binarnym kodem maszynowym (czyli na przykład zestawem instrukcji dla rodziny procesorów Intel czy ARM). Kod bajtowy jest charakterystyczną dla Pythona reprezentacją programu.

Z powyższych przyczyn część kodu napisanego w Pythonie może nie działać tak szybko, jak kod w językach C czy C++, o czym wspominaliśmy w rozdziale 1. To maszyna wirtualna Pythona, a nie procesor, musi zinterpretować kod bajtowy, a wykonanie pojedynczej instrukcji kodu bajtowego wymaga wykonania znacznie większej liczby operacji niż wykonanie pojedynczej instrukcji procesora. Z drugiej strony, w przeciwieństwie do klasycznych interpreterów nadal istnieje tutaj etap komplikacji wewnętrznej — Python nie musi ciągle analizować i ponownie przetwarzając każdej instrukcji z kodu źródłowego. Rezultat jest taki, że kod napisany w samym Pythonie działa z szybkością znajdującej się gdzieś pomiędzy szybkością tradycyjnych języków komplikowanych a szybkością tradycyjnych języków interpretowanych. Więcej informacji dotyczących wydajności Pythona znajdziesz w rozdziale 1.

Wpływ na proces programowania

Inną konsekwencją modelu wykonawczego Pythona jest to, że tak naprawdę nie istnieje rozróżnienie pomiędzy środowiskiem programistycznym a środowiskiem wykonawczym. W skrócie mówiąc, systemy odpowiedzialne za komplikację kodu źródłowego oraz jego wykonywanie są w rzeczywistości jednym i tym samym. To podobieństwo może być nieco bardziej istotne dla osób, które znają już tradycyjne języki komplikowane. W Pythonie komplikator jest zawsze obecny w czasie wykonywania i jest częścią systemu uruchamiającego programy.

To wszystko sprawia, że cykl tworzenia oprogramowania jest o wiele krótszy. Nie ma konieczności wcześniejszego komplikowania i łączenia przed rozpoczęciem wykonywania programu — wystarczy wpisać kod i możemy go od razu wykonać. Powoduje to, że Python to język bardzo dynamiczny — możliwe (i często bardzo wygodne) jest tworzenie i wykonywanie nowych programów Pythona z poziomu innego programu napisanego w tym języku. Wbudowane funkcje eval oraz exec mogą pobierać i wykonywać ciągi znaków zawierające kod Pythona. Taka struktura jest również przyczyną częstego wykorzystywania Pythona w dostosowywaniu produktów do własnych potrzeb — ponieważ kod Pythona może być zmieniany w locie, użytkownicy mogą modyfikować poszczególne części systemu na miejscu bez konieczności ponownego komplikowania kodu całej aplikacji.

Na bardziej podstawowym poziomie należy pamiętać, że w Pythonie istnieje tak naprawdę tylko faza *wykonywania* — nie ma początkowej fazy komplikacji, a wszystko dzieje się w trakcie działania programu. Ten model obejmuje również takie operacje, jak tworzenie funkcji czy klas oraz dołączanie modułów. W językach bardziej statycznych takie zdarzenia występują przed rozpoczęciem wykonywania programu, jednak w Pythonie mają miejsce dopiero w czasie jego działania. Jak się niebawem przekonasz, rezultatem jest bardziej dynamiczne środowisko programistyczne niż to, do którego niektórzy użytkownicy mogą być przyzwyczajeni.

Warianty modeli wykonywania

Przed przejściem do omawiania kolejnych kwestii warto wspomnieć o tym, że opisany tutaj wewnętrzny model wykonywania jest charakterystyczny dla dzisiejszej standardowej implementacji Pythona, jednak nie jest tak naprawdę wymaganiem ze strony samego języka. Z tego powodu z czasem model ten może się zmieniać. Już teraz istnieje kilka implementacji Pythona modyfikujących proces przedstawiony na rysunku 2.2. Omówmy zatem najważniejsze z tych wariantów.

Alternatywne implementacje Pythona

Gdy pracowałem nad tym wydaniem książki, istniało co najmniej pięć różnych implementacji języka Python — *CPython*, *Jython*, *IronPython*, *Stackless* i *PyPy*. Choć istnieje pomiędzy nimi wiele wzajemnych inspiracji i powiązań, każda z nich jest oddzielnym systemem oprogramowania, z własnymi zespołami deweloperów i bazą użytkowników. Do innych potencjalnych kandydatów do tego miana należą pakiety *Cython* i *Shed Skin*, ale będą one omawiane później jako narzędzia optymalizacyjne, ponieważ nie implementują standardowego języka Python (pierwszy z nich jest mieszanką Pythona i języka C, a drugi wykorzystuje niejawne typowanie statyczne).

Mówiąc w skrócie, *CPython* jest implementacją standardową, z której korzysta większość użytkowników (prawdopodobnie również i Ty). Jest to również wersja, która została użyta w tej książce, chociaż podstawowe konstrukcje języka Python w wymienionych wcześniej implementacjach alternatywnych są niemal identyczne. Każdy z wymienionych pakietów ma swoje określone cele i przeznaczenie, choć często mogłyby także służyć jako zamienniki dla *CPythona*. Wszystkie implementują ten sam język Python, ale wykonują programy na różne sposoby.

Na przykład *PyPy* jest dobrym zamiennikiem dla *CPythona*, potrafiącym znacznie szybciej wykonywać większość programów. Podobnie *Jython* i *IronPython* są całkowicie niezależnymi implementacjami Pythona, które komplikują kod źródłowy dla różnych architektur środowiska wykonawczego, zapewniając bezpośredni dostęp do komponentów Java i .NET. Uzyskanie dostępu do oprogramowania Java i .NET możliwe jest również z poziomu programów napisanych dla wersji *CPython* — istnieją na przykład biblioteki *JPyte* i *Python for .NET*, umożliwiające wywoływanie komponentów Java i .NET z poziomu standardowego kodu *CPython*. Wersje *Jython* i *IronPython* oferują bardziej kompletne rozwiązania, udostępniając pełne implementacje języka Python.

Oto krótki przegląd najważniejszych dostępnych obecnie implementacji Pythona.

CPython — standard

Oryginalna, standardowa implementacja języka Python jest zwykle nazywana *CPython*, zwłaszcza gdy chcemy ją porównać z innymi wydaniami (czy po prostu „zwykłym Pythonem”). Jej nazwa wzięła się od tego, że została napisana w przenośnym kodzie w języku ANSI C. To jest właśnie ten Python, którego możemy pobrać ze strony <http://www.python.org>, który jest

udostępniany w dystrybucjach ActivePython i Enthought, czy który instalowany jest automatycznie na większości komputerów z systemami Linux oraz macOS. Jeżeli na Twoim komputerze masz już zainstalowaną jakąś wersję Pythona, najprawdopodobniej będzie to właśnie CPython — o ile oczywiście firma czy organizacja, w której pracujesz, nie wykorzystuje Pythona w jakiś inny, wyspecjalizowany sposób.

Jeżeli nie tworzysz w Pythonie skryptów dla aplikacji napisanych w Javie czy .NET ani nie musisz korzystać z nowych możliwości, jakie dają dystrybucje takie jak Stackless czy PyPy, to najprawdopodobniej używasz standardowej dystrybucji CPython. Ponieważ jest to referencyjna implementacja języka Python, to zwykle jest też najszybsza, najbardziej kompletna i ma największe możliwości w porównaniu z rozwiązaniami alternatywnymi. Rysunek 2.2 odzwierciedla architekturę wykonywania tej implementacji.

Jython — Python dla języka Java

Dystrybucja Jython (wcześniej znana jako JPython) jest alternatywną implementacją języka Python skierowaną na integrację z językiem programowania Java. Jython składa się z klas Javy komplikujących kod źródłowy Pythona na kod bajtowy Javy, a następnie kierujących wynikowy kod bajtowy do maszyny wirtualnej Javy (Java Virtual Machine, JVM). Programiści nadal zapisują kod Pythona w plikach tekstowych z rozszerzeniem `.py`, natomiast Jython zastępuje środkową i prawą część rysunku 2.2 odpowiednikami z języka Java.

Celem dystrybucji Jython jest umożliwienie tworzenia w Pythonie skryptów dla aplikacji napisanych w Javie w podobny sposób, jak CPython pozwala na pisanie skryptów dla aplikacji w językach C czy C++. Integracja tego systemu z Javą jest niezwykle płynna. Ponieważ kod napisany w Pythonie przekładany jest na kod bajtowy Javy, wygląda on i zachowuje się w czasie działania zupełnie jak prawdziwy program napisany w Javie. Skrypty Jytthona mogą służyć jako apety webowe oraz serwety czy tworzyć graficzne interfejsy użytkownika (GUI) oparte na Javie. Co więcej, Jython posiada również moduły pozwalające na importowanie i wykorzystywanie klas Javy tak jakby były napisane w Pythonie, a także używanie kodu Java do uruchamiania osadzonego kodu napisanego w języku Python. Ponieważ jednak Jython jest wolniejszy i ma mniej możliwości od standardowej implementacji CPython, zazwyczaj postrzegany jest jako narzędzie interesujące głównie programistów Javy szukających języka skryptowego, który będzie mógł być frontendem dla kodu napisanego w Javie. Więcej szczegółowych informacji na ten temat znajdziesz na stronie projektu Jython (<https://www.jython.org/>).

IronPython — Python dla .NET

Iron-Python to implementacja Pythona nowsza niż CPython i Jython, która została zaprojektowana z myślą o umożliwieniu integracji programów napisanych w Pythonie z aplikacjami stworzonymi dla platformy .NET firmy Microsoft przeznaczonej dla systemu Windows, a także dla platformy Mono, będącej otwartoźródłowym odpowiednikiem .NET dla systemu Linux. Platforma .NET wraz ze swoim środowiskiem uruchomieniowym z językiem C# zaprojektowana jest w taki sposób, aby stanowić niezależną od języka warstwę komunikacji obiektowej, wzorowaną na wcześniejszym modelu COM firmy Microsoft. IronPython pozwala programom napisanym w Pythonie działać jako komponenty klienta, jak i serwera, dostępne z innych języków środowiska .NET oraz wykorzystujące inne technologie .NET, takie jak framework *Silverlight* z poziomu kodu Pythona.

Ze sposobu implementacji IronPython jest bardzo podobny do dystrybucji Jython (tak naprawdę został opracowany przez tę samą osobę) — zastępuje on środkową i prawą część rysunku 2.2 odpowiednikami pozwalającymi na wykonywanie kodu w środowisku .NET. Podobnie jak Jython, IronPython ma szczególne przeznaczenie — jest interesujący przede wszystkim dla programistów zajmujących się integracją Pythona z komponentami .NET. Wcześniej opracowany przez firmę Microsoft, a obecnie funkcjonujący jako projekt *open source*, IronPython może korzystać z wybranych narzędzi optymalizacyjnych, pozwalających na polepszenie jego wydajności. Więcej szczegółowych informacji na ten temat znajdziesz na stronie <http://ironpython.net> oraz innych stronach poświęconych temu językowi.

Stackless: Python dla programowania współbieżnego

Inne dystrybucje języka Python mają jeszcze bardziej ukierunkowane cele. Na przykład dystrybucja *Stackless Python* jest rozszerzoną wersją i reimplementacją standardowego języka CPython, zorientowaną na współbieżność. Ponieważ w tej implementacji stany nie są zapisywane na stosie wywołań języka C, Stackless Python ułatwia przenoszenie programów do systemów wyposażonych w stosy o małych rozmiarach, zapewnia wydajne przetwarzanie w systemach wieloprocesorowych i sprzyja nowatorskim strukturom programowania, takim jak tzw. współprограмy (ang. *coroutines*).

Oprócz wielu innych, ciekawych komponentów, Stackless dodaje do Pythona tzw. *mikrowątki* (ang. *microthreads*), które są wydajną i lekką alternatywą dla standardowych narzędzi wielozadaniowych Pythona, takich jak wątki i procesy. Stackless oferuje lepszą strukturę programów, bardziej czytelny kod i zwiększoną wydajność programistów. Firma CCP Games, twórca *EVE Online*, jest powszechnie znanym użytkownikiem dystrybucji Stackless Python i może być znakomitym przykładem historii sukcesu w świecie Pythona. Więcej szczegółowych informacji na temat tej dystrybucji znajdziesz na stronie <http://stackless.com>.

PyPy — Python dla szybkości i wydajności

Dystrybucja PyPy to kolejna reimplementacja standardowego CPythona, skoncentrowana na *wydajności*. Jest to szybka implementacja Pythona z kompilatorem JIT (ang. *just-in-time*), która m.in. oferuje narzędzia do obsługi „piaskownicy” (ang. *sandbox*), pozwalającej na uruchamianie niezaufanego kodu w bezpiecznym środowisku; posiada ona także wbudowaną obsługę mikrowątków znanych z implementacji Stackless, pozwalających na uzyskiwanie bardzo efektywnej współbieżności.

PyPy jest następcą oryginalnego kompilatora *Psyco JIT* (opisanego w jednym z kolejnych podrozdziałów) i uzupełnia go o pełną implementację Pythona zbudowaną z myślą o szybkości. JIT jest tak naprawdę rozszerzeniem maszyny wirtualnej PVM (czyli skrajnego, prawego elementu na rysunku 2.2), która tłumaczy fragmenty kodu bajtowego na binarny kod maszynowy w celu szybszego wykonania. Operacja taka odbywa się w czasie *działania* programu, a nie podczas komplikacji wstępnej. PyPy śledzi rodzaje przetwarzanych obiektów, dzięki czemu jest w stanie dynamicznie tworzyć kod maszynowy zoptymalizowany dla określonych *typów danych*. Zastępując w ten sposób fragmenty kodu bajtowego, PyPy powoduje, że program działa coraz szybciej. Warto również zauważyć, że programy napisane i uruchamiane w PyPy z reguły mają mniejsze zapotrzebowanie na pamięć operacyjną niż ich „klasyczne” odpowiedniki.

W czasie kiedy powstawała ta książka, PyPy obsługiwał kod języka Python 2.7 (ale jeszcze nie w wersji 3.x) i działał na platformach Intel x86 (IA-32) i x86_64 (w tym Windows, Linux i najnowszych systemach macOS), a obsługa ARM i PPC była w przygotowaniu. Większość kodu napisanego dla CPython będzie w PyPy działać poprawnie, choć moduły rozszerzeń C zazwyczaj muszą być rekompilowane. W PyPy występują pewne niewielkie, ale subtelne różnice językowe, w tym w semantyce poleceń odpowiadających za czyszczenie pamięci i usuwanie niepotrzebnych plików (ang. *garbage collection*), która eliminuje niektóre typowe problemy kodowania. Na przykład brak mechanizmu zliczania odwołań może powodować, że pliki tymczasowe nie będą zamknięte, a powiązane z nimi bufore wyjściowe nie będą natychmiast opróżniane, co w niektórych przypadkach może wymagać ręcznego zamknięcia takich plików.

W zamian jednak Twój kod może działać znacznie szybciej. W wielu programach testowych (dostępnych na <http://speed.pypy.org>) PyPy jest szybszy prawie sześć razy niż klasyczny CPython. W niektórych przypadkach możliwość wykorzystania dynamicznej optymalizacji wykonywanego programu może sprawić, że kod Pythona będzie tak szybki jak kod napisany w C, a czasami nawet szybszy. Jest to szczególnie prawdziwe w przypadku programów mocno zalgorytmizowanych lub przeprowadzających intensywne obliczenia numeryczne, które w przeciwnym razie mogłyby zostać napisane w języku C.

Na przykład, w jednym prostym teście wydajności (ang. *benchmark*), który pokażemy w rozdziale 21., PyPy osiąga dzisiaj *dziesięciokrotnie* większą szybkość niż CPython 2.7 i jest ponad *stukrotnie* szybszy niż CPython 3.x. Chociaż inne wyniki innych testów mogą się różnić, takie przyspieszenie może być istotną zaletą w wielu dziedzinach, być może nawet większą niż nowatorskie funkcje języka. Równie ważne jest to, że zarządzanie pamięcią w PyPy jest również zoptymalizowane — w przypadku jednego z opublikowanych testów, program napisany w PyPy poradził sobie z nim w 10,3 sekundy, zużywając 247 MB pamięci, podczas gdy CPython potrzebował do tego 684 MB pamięci i aż 89 sekund.

Zestaw narzędzi PyPy jest również na tyle uniwersalny, że potrafi obsługiwać dodatkowe języki, takie jak m.in. *Pyrolog*, interpreter języka Prolog napisany w Pythonie przy użyciu translatora PyPy. Więcej szczegółowych informacji na ten temat znajdziesz na stronie projektu PyPy, pod adresem <http://pypy.org>; owocne może się okazać również zwykłe zapytanie w wyszukiwarce sieciowej. Aktualne informacje o zagadnieniach związanych z wydajnością PyPy znajdziesz na stronie <http://www.pypy.org/performance.html>.

	<p>Niedługo po tym, jak to napisałem, na stronie projektu PyPy pojawiła się wersja 2.0 beta, w której dodane zostało wsparcie dla procesora ARM (ale wersja ta nadal obsługuje tylko Pythona 2.x). Według załączonych informacji o wersji 2.0 beta:</p> <p>„PyPy jest bardzo zgodnym interpreterem Pythona i niemal znakomitym zamiennikiem CPythona 2.7.3. Jest szybki (zobacz http://speed.pypy.org) dzięki zintegrowanemu kompilatorowi JIT. To wydanie obsługuje maszyny z procesorami o architekturze x86 z systemami Linux 32/64, Mac OS X 64 lub Windows 32. Oprócz tego obsługuje również maszyny z procesorami ARM, działające pod kontrolą systemu Linux”.</p> <p>Powyższe oświadczenie wydaje się być dokładne. Używając narzędzi do pomiaru czasu, które zaprezentujemy w rozdziale 21., mogę stwierdzić, że w wielu testach, które przeprowadziłem, PyPy był często o rząd wielkości (10x) szybszy niż CPython 2.x i 3.x, a czasem nawet jeszcze szybszy. Działo się tak pomimo faktu, że na moim komputerze używałem 32-bitowej wersji PyPy, podczas gdy CPython był w teoretycznie szybszej wersji 64-bitowej.</p> <p>Oczywiście jednak jedynym kryterium, które naprawdę ma znaczenie, jest Twój własny kod i zdarzają się przypadki, w których to CPython wygrywa wyścig — przykładowo, w obecnej wersji iteratory PyPy mogą działać nieco wolniej niż w CPythonie. Mimo to, biorąc pod uwagę, że PyPy jest zorientowany na osiąganie maksymalnej wydajności kosztem pewnych mutacji językowych, co widać zwłaszcza w przypadku obliczeń numerycznych, wielu użytkowników postrzega dzisiaj PyPy jako ważną ścieżkę rozwoju dla Pythona. Jeżeli piszesz programy, które intensywnie wykorzystują procesor komputera i inne jego zasoby, PyPy z pewnością zasługuje na Twoją uwagę.</p>
--	---

Narzędzia do optymalizacji działania programu

CPython, podobnie jak większość alternatywnych dystrybucji, opisywanych w poprzednich sekcjach, implementuje język Python w podobny sposób — kod źródłowy jest kompilowany do postaci kodu bajtowego, który następnie jest wykonywany na odpowiedniej maszynie wirtualnej. Inne systemy, takie jak Cython hybrid, translator Shedskin C++ czy kompilatory JIT próbują zamiast tego optymalizować podstawowy model wykonywania. Na tym etapie znajomość tych narzędzi nie jest jeszcze konieczna, jednak krótkie spojrzenie na ich miejsce w modelu wykonywania może pomóc Ci w lepszym zrozumieniu sposobu działania tego modelu.

Cython: hybryda Pythona/C

Dystrybucja Cython (oparta na projekcie *Pyrex*) jest językiem hybrydowym, który łączy kod Pythona z możliwością wywoływania funkcji języka C i używania jego deklaracji typów dla zmiennych, parametrów i atrybutów klas. Kod Cythona można skompilować do kodu w języku C, który używa interfejsu API Python/C, a następnie może zostać skompilowany całkowicie. Chociaż Cython nie jest w pełni kompatybilny ze standardowym Pythonem, może być przydatny zarówno do pakowania zewnętrznych bibliotek C, jak i do kodowania wydajnych rozszerzeń języka C dla Pythona. Więcej szczegółowych informacji na ten temat znajdziesz na stronie projektu Cython, dostępnej pod adresem <http://cython.org>.

Shed Skin: translator języka Python na C ++

Shed Skin to nowy, intensywnie rozwijany system, który przyjmuje nieco inne podejście do wykonywania programów w języku Python — próbuje tłumaczyć kod źródłowy Pythona na kod C++, który następnie jest komplikowany do postaci kodu maszynowego przez kompilator C++. Dzięki takiemu rozwiązaniu Shed Skin reprezentuje niezależne od platformy podejście do uruchamiania kodu napisanego w języku Python. Shed Skin jest nadal aktywnie rozwijany. Obecnie obsługuje kod Pythona w wersjach od 2.4 do 2.6 i wymusza stosowanie w programach niejawnego, statycznego typowania danych, co jest typowe dla większości programów. Technicznie rzecz biorąc, Shed Skin nie jest jednak normalnym Pythonem, więc nie będziemy tutaj wchodzić w szczegóły. Wstępne analizy pokazują, że ma on potencjał, by przewyższyć zarówno standardowe rozszerzenia Pythona, jak i inne kompilatory, takie jak Psyco, pod względem szybkości wykonywania. Więcej szczegółowych informacji na ten temat znajdziesz na stronie internetowej projektu.

Psyco — oryginalny kompilator JIT

Psyco nie jest kolejną implementacją Pythona, ale raczej komponentem rozszerzającym model wykonywania kodu bajtowego, który sprawia, że programy mogą działać szybciej. Niestety, obecnie Psyco jest projektem *wygasłym* — nadal możemy go pobierać, ale przestał być rozwijany i aktualizowany. Rozwiązania wykorzystywane w Psyco zostały włączone do bardziej kompletnej dystrybucji PyPy, którą opisywaliśmy wcześniej. Znaczenie rozwiązań opracowanych i zastosowanych w projekcie Psyco jest jednak na tyle duże, że warto o nich powiedzieć kilka słów.

W kategoriach rysunku 2.2 Psyco jest ulepszeniem maszyny wirtualnej Pythona, które w czasie działania programu zbiera informacje o typach danych i wykorzystuje je do tłumaczenia fragmentów kodu bajtowego programu na prawdziwy kod maszynowy, tak aby przyspieszyć jego wykonywanie. Psyco wykonuje taką operację bez konieczności wprowadzania zmian w kodzie czy osobnego etapu kompilacji w czasie tworzenia programu.

W skrócie, kiedy program zostaje uruchomiony, Psyco zbiera informacje o typach obiektów, jakie są w nim przekazywane. Zebrane informacje są wykorzystywane do wygenerowania bardzo wydajnego kodu maszynowego dostosowanego do takich obiektów. Po wygenerowaniu kod maszynowy zastępuje odpowiadającą mu część oryginalnego kodu bajtowego, co przyspiesza ogólną szybkość wykonywania programu. Dzięki kompilatorowi Psyco po uruchomieniu programu w miarę upływu czasu staje się coraz szybszy. W idealnych przypadkach część kodu w Pythonie może dzięki Psyco działać z szybkością niemal porównywalną do skompilowanego kodu w języku C.

Ponieważ proces przekładania kodu bajtowego na maszynowy odbywa się w czasie działania programu, Psyco znany jest jako *kompilator JIT* (ang. *just-in-time*). Psyco różni się jednak nieco od kompilatorów JIT znanych z języka Java. Psyco jest *wyspecjalizowanym kompilatorem JIT* — generuje kod maszynowy dopasowany do typów danych, jakie wykorzystuje dany program. Jeżeli na przykład część programu wykorzystuje różne typy danych w różnym czasie, Psyco może wygenerować różne wersje kodu maszynowego obsługujące każdą z takich kombinacji typów.

Zostało udowodnione, że kompilator Psyco jest w stanie znaczco zwiększyć szybkość działania programów napisanych w Pythonie. Według informacji na stronie internetowej projektu, Psyco

zapewnia „przyspieszenie działania programu od dwóch do nawet stu razy, zazwyczaj około czterech razy, bez modyfikowania interpretera Pythona i bez modyfikowania kodu źródłowego, wyłącznie dzięki zastosowaniu dynamicznie ładowanego modułu rozszerzenia w języku C”. Co równie ważne, największe przyspieszenie można zaobserwować dla kodu algorytmicznego napisanego w czystym Pythonie — czyli właśnie tego rodzaju kodu, jaki normalnie przenosi się do języka C w celu jego optymalizacji. Więcej szczegółowych informacji na ten temat znajdziesz na stronie projektu Psyco oraz w opisie PyPy, będącego nieformalnym następcą kompilatora Psyco.

Zamrożone pliki binarne

Czasami, kiedy użytkownicy pytają o „prawdziwy” kompilator Pythona, tak naprawdę szukają prostego sposobu generowania samodzielnych, wykonywalnych plików binarnych dla programów napisanych w Pythonie. Bardziej chodzi tu o tworzenie pakietów i udostępnianie ich niż o modyfikację procesu wykonywania, jednak w pewien sposób te dwie koncepcje są ze sobą powiązane. Dzięki pomocy dodatkowych narzędzi, które można pobrać z internetu, można zmienić programy napisane w Pythonie w prawdziwe pliki wykonywalne, znane w świecie Pythona jako *zamrożone pliki binarne* (ang. *frozen binaries*) — takie pliki można uruchamiać na różnych komputerach bez konieczności instalowania na nich interpretera języka Python.

Zamrożone pliki binarne obejmują kod bajtowy plików programu wraz z maszyną wirtualną Pythona (interpreterem) i wszelkimi plikami pomocniczymi Pythona, jakich potrzebuje program, połączonymi w jeden pakiet. Istnieje kilka odmian tej koncepcji, jednak rezultatem końcowym najczęściej może być pojedynczy binarny program wykonywalny (na przykład plik .exe w przypadku systemu Windows), który można w łatwy sposób udostępnić klientom. Na rysunku 2.2 wyglądałoby to tak, jakby dwa elementy z prawej strony — kod bajtowy i PVM — zostały połączone w jeden komponent: zamrożony plik binarny.

Obecnie istnieje wiele programów mogących generować zamrożone pliki binarne. Poszczególne produkty różnią się od siebie zarówno obsługiwany platformami, jak i funkcjonalnością: *py2exe* przeznaczony jest tylko dla systemu Windows, *PyInstaller*, który jest podobny do *py2exe*, ale działa również na systemach Linux oraz macOS i potrafi generować samoinstalujące się pliki binarne; *py2app* do tworzenia aplikacji macOS; *freeze*, czyli oryginalne narzędzie instalowane razem z interpreterem Pythona oraz *cx_freeze*, który potrafi kompilować programy napisane w języku Python 3.x, przeznaczone dla różnych platform. Większość tych narzędzi trzeba pobierać niezależnie od Pythona, ale są one darmowe.

Wymienione narzędzia są ciągle rozwijane, dlatego warto śledzić ich postępy na stronie <http://www.python.org> lub za pośrednictwem ulubionej wyszukiarki internetowej. Aby pokazać, jaki może być zakres przydatności tych narzędzi, wystarczy wspomnieć, że *py2exe* może „zamrozić” programy wykorzystujące biblioteki *tkinter*, *PMW*, *wxPython* oraz *PyGTK GUI*, a także programy korzystające z zestawu narzędzi programistycznych *pygame* czy biblioteki *win32com*.

Zamrożone pliki binarne nie są tym samym co wyniki działania prawdziwego kompilatora — nadal wykonują one kod bajtowy za pomocą maszyny wirtualnej. Z tego względu, z wyjątkiem nieco szybszego uruchamiania, zamrożone programy działają z szybkością podobną do oryginalnych plików źródłowych. Zamrożone pliki binarne zazwyczaj nie są małe (zawierają maszynę wirtualną Pythona), ale nie są też szczególnie duże — jak na dzisiejsze standardy. Ponieważ interpreter Pythona jest osadzany w zamrożonych plikach binarnych, nie musi on być zainstalowany na maszynie docelowej, by móc wykonać program. Co więcej, ponieważ kod źródłowy programu jest osadzony w zamrożonym pakiecie, jest bardziej efektywnie ukryty przed użytkownikiem.

Taki schemat wykorzystujący pojedyncze pliki (pakiety) jest szczególnie atrakcyjny dla twórców oprogramowania komercyjnego. Program interfejsu użytkownika oparty na zestawie narzędzi Tkinter można na przykład zamrozić w pliku wykonywalnym i udostępnić jako samodzielny

program na płycie CD czy w internecie. Użytkownicy nie muszą instalować samego Pythona (czy w ogóle wiedzieć o jego istnieniu), żeby uruchomić udostępniany w ten sposób program.

Przyszłe możliwości?

Na koniec warto wspomnieć, że omówiony tutaj model wykonywania Pythona jest tak naprawdę charakterystyczny dla bieżącej implementacji Pythona, a nie dla samego języka. Całkiem możliwe, że jeszcze w czasie, kiedy niniejsza książka będzie dostępna na półkach w księgarniach, pojawi się pełnowymiarowy, tradycyjny kompilator przekładający kod źródłowy Pythona na kod maszynowy (choć fakt, że nie stało się tak w czasie ostatnich dwóch dziesięcioleci wskazuje, że prawdopodobieństwo takiego zdarzenia jest niewielkie).

W przyszłości mogą pojawić się również nowe formaty kodu bajtowego czy warianty implementacyjne. Na przykład:

- Celem projektu *Parrot* jest opracowanie ujednoliconego, wspólnego formatu kodu bajtowego, maszyny wirtualnej i technik optymalizacyjnych dla różnych języków programowania, włącznie z Pythonem. Własna maszyna wirtualna Pythona (PVM) wykonuje kod napisany w tym języku bardziej efektywnie niż Parrot, (jak to zostało pokazane na jednej z konferencji poświęconych oprogramowaniu), ale nie jest do końca jasne, w jaki sposób Parrot będzie ewoluować w odniesieniu do Pythona. Szczegółowe informacje można znaleźć na stronie projektu <http://parrot.org> lub na innych stronach w internecie.
- *Unladen Swallow*, zawieszony już od kilku lat projekt open source opracowany przez inżynierów Google, miał na celu co najmniej pięciokrotne przyspieszenie standardowego Pythona tak, aby mógł zastępować język C w wielu zastosowaniach. Projekt był optymalizowanym rozgałęzieniem implementacji CPython (a konkretnie wersji Python 2.6), która w zamierzeniach ma być w pełni zgodna z CPythonem, a jednocześnie znacznie szybsza, dzięki dodaniu do standardowej implementacji kompilatora JIT. Obecnie wydaje się, że projekt Unladen Swallow został zakończony (lub też, nawiązując do jednego ze słynnych skrecczy Monty Pythona, „podążył drogą papugi Norwegian Blue”). Nie zmienia to jednak w niczym faktu, że rozwiązania opracowane przez programistów Unladen Swallow z pewnością mogą być wykorzystane w innych projektach; więcej szczegółowych informacji na ten temat znajdziesz w internecie.

Choć przyszłe schematy implementacyjne mogą w jakimś stopniu zmienić strukturę wykonywania Pythona, wydaje się, że kompilator kodu bajtowego przez jakiś czas pozostanie jeszcze standardem. Przenośność i elastyczność działania kodu bajtowego są ważnymi cechami wielu implementacji Pythona. Co więcej, sztuczne ograniczanie typów danych dla ułatwienia komplikacji statycznej najprawdopodobniej zniszczyłoby elastyczność, zwięzłość, prostotę i ogólnego ducha Pythona. Ze względu na wysoce dynamiczną naturę Pythona jeszcze wiele jego przyszłych implementacji będzie z pewnością zawierało dziedzictwo dzisiejszej maszyny wirtualnej Pythona.

Podsumowanie rozdziału

W niniejszym rozdziale omówiono model wykonywania obecny w Pythonie (czyli to, jak Python wykonuje programy), a także popularne odmiany tego modelu (na przykład kompilatory JIT). Choć do pisania skryptów w tym języku nie jest konieczne poznanie wewnętrznych mechanizmów Pythona, zapoznanie się z tematami poruszonymi w niniejszym rozdziale pomoże nam zrozumieć, jak działają nasze programy, kiedy już zaczniemy je tworzyć. W kolejnym rozdziale zaczniesz już pisać i uruchamiać swoje pierwsze programy. Najpierw jednak czas na kolejny quiz, podsumowujący zagadnienia omawiane w tym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Co to jest interpreter Pythona?
2. Co to jest kod źródłowy?
3. Co to jest kod bajtowy?
4. Co to jest PVM?
5. Podaj nazwy co najmniej dwóch wariantów standardowego modelu wykonywania Pythona.
6. Czym różnią się od siebie CPython, Jython oraz IronPython?
7. Czym są Stackless i PyPy?

Sprawdź swoją wiedzę — odpowiedzi

1. Interpreter Pythona to program wykonujący programy napisane w języku Python.
2. Kod źródłowy to ciągi instrukcji, które wpisujesz, aby utworzyć program. Kod źródłowy jest zapisywany w postaci tekstuowej w plikach, których nazwy zazwyczaj kończą się rozszerzeniem `.py`.
3. Kod bajtowy to niskopoziomowa postać programu powstająca po skompilowaniu przez Pythona. Python automatycznie przechowuje kod bajtowy w plikach z rozszerzeniem `.pyc`.
4. PVM to maszyna wirtualna Pythona (ang. *Python Virtual Machine*) — silnik wykonawczy Pythona, który interpretuje nasz skompilowany kod.
5. Psyco, Shed Skin i zamrożone pliki binarne to różne alternatywy dla modelu wykonywania Pythona. Warto zauważyć, że alternatywne implementacje Pythona wymienione w kolejnych dwóch odpowiedziach, również modyfikują w pewien sposób model wykonywania, zastępując kod bajtowy i maszyny wirtualne lub dodając swoje narzędzia i kompilatory JIT.
6. CPython to standardowa implementacja języka Python. Dystrybucje Jython oraz IronPython pozwalają na wykorzystywanie programów napisanych w języku Python w środowiskach odpowiednio Java i .NET; są alternatywnymi kompilatorami Pythona.
7. Stackless jest ulepszoną wersją Pythona ukierunkowaną na współpracę, a PyPy jest reimplementacją Pythona ukierunkowaną na szybkość działania. PyPy jest również następcą kompilatora Psyco i wykorzystuje niektóre rozwiązania JIT, które zostały opracowane na potrzeby projektu Psyco.

Rozdział 3. Jak wykonuje się programy?

Nadszedł już czas, aby w końcu uruchomić jakiś kod. Skoro wiemy już nieco o modelu wykonywania programów, możemy wreszcie zabrać się za prawdziwe programowanie w Pythonie. Od teraz zakładam, że masz już na swoim komputerze zainstalowanego Pythona. Jeżeli tak nie jest, powinieneś wrócić na początek poprzedniego rozdziału oraz zatrzymać się tutaj, aż skończysz instalację i konfigurację Pythona.

Istnieje wiele różnych sposobów nakazania Pythonowi, by wykonał wpisywany przez Ciebie kod. W tym rozdziale omówimy wszystkie techniki uruchamiania programów będące obecnie w powszechnym użyciu. Nauczysz się, w jaki sposób *interaktywnie* wpisywać kod i jak zapisywać ten kod w *plikach*, tak by można je było uruchamiać w dowolnej chwili na wiele sposobów: z poziomu wiersza poleceń, klikając ikonę pliku, importując moduły, wywołując funkcję `exec` czy wybierając odpowiednie polecenie z menu graficznych interfejsów użytkownika, takich jak w środowisku IDLE.

Podobnie jak w poprzednim rozdziale, jeżeli masz już jakieś doświadczenie w programowaniu i chcesz po prostu zacząć zgłębiać tajniki Pythona, możesz szybko przekartkować ten rozdział i przejść do rozdziału 4. Nie pomijaj jednak części poświęconych przygotowaniom i konwencjom, przeglądowi technik debuggowania czy pierwszego spojrzenia na importowanie modułów — są to zagadnienia niezbędne do zrozumienia architektury programów Pythona, do którego powróćmy dopiero w dalszej części rozdziału. Zachęcam również do zapoznania się z sekcjami dotyczącymi interfejsów środowiska IDLE (ang. *Integrated Development and Learning Environment*) i innych zintegrowanych środowisk programistycznych (IDE — ang. *Integrated Development Environment*), abyś wiedział jakie narzędzia są dostępne, gdy zaczniesz tworzyć bardziej zaawansowane programy w języku Python.

Interaktywny wiersz poleceń

W tej sekcji rozpoczynamy od przedstawienia podstawowych zagadnień związanych z interaktywnym uruchamianiem poleceń Pythona. Ponieważ jest to nasze pierwsze spojrzenie na uruchamianie kodu, omówimy tutaj również kilka elementów wstępnych, takich jak konfigurowanie katalogu roboczego i ścieżki systemowej, więc jeżeli są to dla Ciebie nowe tematy, powinieneś zacząć czytanie od tej właśnie sekcji. W tej części opiszemy również niektóre konwencje stosowane w książce, więc większość Czytelników powinna tutaj przynajmniej zatrzymać się.

Uruchamianie sesji interaktywnej

Chyba najprostszym sposobem wykonywania programów w Pythonie jest wpisywanie ich bezpośrednio z poziomu *interaktywnego wiersza poleceń* (powłoki) Pythona. Powłokę można uruchomić na wiele różnych sposobów — na przykład w IDE czy z konsoli systemowej. Zakładając, że interpreter zainstalowany jest w naszym systemie jako program wykonywalny, najbardziej niezależnym od platformy sposobem rozpoczęcia interaktywnej sesji interpretera

jest zazwyczaj wpisanie słowa `python` w wierszu poleceń systemu operacyjnego, bez żadnych argumentów. Na przykład:

```
% python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit
...
Type "help", "copyright", "credits" or "license" for more information.

>>> ^Z
```

Wpisanie słowa `python` w wierszu poleceń powłoki, tak jak to zostało pokazane powyżej, rozpoczyna interaktywną sesję Pythona (znak % na początku kodu reprezentuje w naszej książce znak zachęty systemu i nie jest elementem, który należy wpisywać). Aby zakończyć interaktywną sesję powłoki Pythona, w systemie Windows powinieneś nacisnąć kombinację klawiszy `Ctrl+Z`; jeżeli pracujesz w Uniksie i podobnych systemach, naciśnij kombinację klawiszy `Ctrl+D`.

Określenie *wiersz poleceń powłoki* w dość ogólny sposób odnosi się do powłoki systemu operacyjnego, choć metody uruchamiania powłoki mogą się różnić dla poszczególnych platform:

- W systemie Windows słowo `python` można wpisać w oknie konsoli systemu Windows (to program o nazwie `cmd.exe`, nazywany w skrócie *wierszem poleceń*); więcej informacji na temat uruchamiania tego programu znajdziesz w ramce „Gdzie w systemie Windows można znaleźć wiersz poleceń?” w dalszej części tego podrozdziału.
- W systemie macOS możesz uruchomić interaktywny interpreter Pythona, wybierając opcję *Aplikacje/Narzędzia/Terminal*, a następnie wpisując polecenie `python` w oknie konsoli, które pojawi się na ekranie.
- W systemie Linux (oraz innych systemach uniksowych) możesz wpisać to polecenie z poziomu powłoki systemu lub w oknie terminala (na przykład w oknie terminala `xterm` lub konsoli powłoki, takiej jak `ksh` czy `csh`).
- Inne systemy mogą wykorzystywać podobne sposoby lub używać do tego celu innych rozwiązań. Na przykład, w urządzeniach przenośnych do uruchomienia sesji interaktywnej wystarczy zazwyczaj kliknięcie ikony Pythona znajdującej się na pulpicie.

Na większości platform można uruchomić interaktywną sesję interpretera Python również na inne sposoby, które nie wymagają wpisywania co prawda żadnych polecień, ale różnią się w zależności od platformy:

- W systemie *Windows 7* i wcześniejszych, oprócz wpisania polecenia `python` w oknie powłoki, możesz również rozpoczęć sesję interaktywną, uruchamiając środowisko IDLE (które zostanie omówione nieco później) lub wybierając polecenie *Python (wiersz poleceń)* z menu *Start*, jak to zostało pokazane na rysunku 2.1 w rozdziale 2. Oba sposoby spowodują uruchomienie interaktywnej konsoli Pythona, tej samej, która jest uruchamiana za pomocą polecenia `python`.
- W systemie *Windows 8* nie ma tradycyjnego przycisku *Start*, ale są inne sposoby, aby dostać się do narzędzi opisanych w poprzednim punkcie, takie jak kafelki, wyszukiwanie, *Eksplorator plików* czy ekran *Aplikacje*. Więcej szczegółowych informacji na ten temat znajdziesz w dodatku A.
- Na innych platformach istnieją podobne sposoby na rozpoczęcie interaktywnej sesji Pythona bez konieczności wpisywania poleczeń z konsoli, ale zbyt różnią się od siebie, aby je tutaj szczegółowo opisywać; więcej informacji na ten temat znajdziesz w dokumentacji danego systemu.

Kiedy na ekranie zobaczysz znak zachęty `>>>`, będzie to oznaczało, że jesteś w interaktywnej sesji interpretera Pythona — możesz tutaj wpisywać dowolne wyrażenia lub polecenia Pythona i

uruchamiać je natychmiast po wpisaniu. Będziemy to robić już za chwilę, ale najpierw musimy omówić jeszcze kilka szczegółów, aby upewnić się, że jesteś w pełni gotowy do pracy.

Gdzie w systemie Windows można znaleźć wiersz poleceń?

Jak można uruchomić wiersz poleceń w systemie Windows? Bardziej doświadczeni użytkownicy systemu Windows zapewne wiedzą, ale początkujący czy deweloperzy pracujący z systemem Unix być może nie, ponieważ w systemie Windows wiersz poleceń nie spełnia tak ważnej roli jak okna terminala czy konsoli w systemach Unix. Oto kilka wskazówek pomagających znaleźć wiersz poleceń, które różnią się nieznacznie w zależności od wersji systemu Windows.

W systemie Windows 7 i wcześniejszych, ikonę wiersza poleceń zazwyczaj można znaleźć w menu Start/Wszystkie programy/Akcesoria. Zamiast jej wyszukiwać, wiersz poleceń można uruchomić po prostu wpisując komendę cmd w oknie dialogowym Start/Uruchom... lub w polu wyszukiwania w menu Start. W razie potrzeby możesz również utworzyć odpowiedni skrót na pulpicie, dzięki czemu będziesz miał szybszy dostęp do wiersza poleceń.

W systemie Windows 8 można uzyskać dostęp do wiersza poleceń w menu otwieranym po kliknięciu prawym przyciskiem myszy w lewym dolnym rogu ekranu; w sekcji System Windows na ekranie Aplikacje, do którego można przejść klikając prawym przyciskiem myszy na ekranie startowym; lub wpisując polecenie cmd w polu wyszukiwania panelu, który włączamy klikając w prawym, górnym rogu ekranu. Prawdopodobnie istnieją jeszcze inne sposoby, a na urządzeniach z ekranami dotykowymi może to wyglądać podobnie. A jeżeli nie chcesz o tym wszystkim pamiętać, możesz po prostu przypiąć ikonę wiersza poleceń do paska zadań na pulpicie, aby mieć do niego łatwy dostęp.

Sposoby uruchamiania wiersza poleceń mogą się zmieniać w kolejnych wersjach systemu, a nawet w zależności od konfiguracji danego komputera i preferencji użytkownika. Starałem się unikać pisania tej książki na bazie systemu Windows, więc nie będę już więcej poruszać tych zagadnień. W razie wątpliwości możesz skorzystać z systemu pomocy Windows (choć jego sposoby uruchamiania mogą się różnić tak samo jak sposoby uruchamiania narzędzi, którym zapewnia pomoc!).

A teraz uwaga dla tych użytkowników systemów Unix, którzy mogli się już poczuć nieco zagubieni — w razie potrzeby możecie skorzystać z pakietu Cygwin, który przynosi w pełni funkcjonalny wiersz poleceń systemu Unix do systemu Windows. Więcej szczegółowych informacji na ten temat można znaleźć w dodatku A.

Ścieżka systemowa

Kiedy w poprzedniej sekcji wpisywaliśmy polecenie python, aby rozpocząć interaktywną sesję, polegaliśmy na fakcie, że system potrafił znaleźć interpreter Pythona w swojej ścieżce wyszukiwania programów. W zależności od wersji Pythona i systemu operacyjnego, jeżeli zmienna środowiskowa powłoki PATH nie zawiera katalogu instalacyjnego Pythona, polecenie python trzeba będzie zastąpić pełną ścieżką do pliku wykonywalnego Pythona zainstalowanego w Twoim komputerze. W systemach Unix, Linux i innych podobnych, często wystarczy wpisanie polecenia /usr/local/bin/python lub /usr/bin/python. W systemie Windows możesz spróbować wpisać C:\Python33\python (w przypadku wersji 3.3).

```
c:\code> c:\python33\python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit
...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

Alternatywnie można przed wpisaniem polecenia `python` wykonać komendę zmieniającą katalog, tak by przejść od razu do katalogu instalacyjnego Pythona (przykładowo, w systemie Windows można użyć polecenia `cd c:\python33`):

```
c:\code> cd c:\python33  
c:\Python33> python  
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit  
...  
Type "help", "copyright", "credits" or "license" for more information.  
>>> ^Z
```

Najprawdopodobniej wcześniej czy później będziesz chciał w końcu ustawić zmienną PATH tak, aby wystarczyło wpisanie prostego polecenia `python`. Jeżeli nie wiesz, czym jest zmienna PATH lub jak ją ustawić, powinieneś zatrzymać się na dodatku A — opisuje on również zmienne środowiskowe, których użycie różni się w zależności od platformy. Znajdziesz tam również opis kilku argumentów wywołania interpretera Pythona, których nie będziemy używać w tej książce. Mała wskazówka dla użytkowników systemu Windows: zatrzymaj się na sekcji *Ustawienia zaawansowane* w sekcji *System* w *Panelu sterowania*. Jeżeli w systemie Windows używasz Pythona 3.3 lub jego nowszych wersji, program instalacyjny może automatycznie wprowadzić odpowiednie zmiany w konfiguracji, jak to wyjaśnia następna sekcja.

Nowe opcje systemu Windows w wersji 3.3: PATH, Launcher

Zarówno poprzednia sekcja, jak i wiele innych z tego rozdziału, w dużej mierze opisuje ogólny stan rzeczy dla wszystkich implementacji Pythona wersji 2.x i 3.x przed wydaniem wersji 3.3. Począwszy od Pythona 3.3, instalator systemu Windows ma możliwość automatycznego dodania katalogu Pythona 3.3 do zmiennej systemowej PATH, o ile taka opcja zostanie włączona w oknie instalatora. Jeżeli używasz tej opcji, nie będziesz musiał wpisywać ścieżki do zmiennej PATH ręcznie ani przechodzić najpierw do katalogu Pythona, aby uruchomić jego powłokę, jak to opisywaliśmy w poprzedniej sekcji. Pamiętaj jednak, aby wybrać tę opcję podczas instalacji, ponieważ jest ona domyślnie wyłączona.

Co więcej, Python 3.3 dla systemu Windows automatycznie instaluje nowe launchery, czyli małe programy uruchomieniowe, które są umieszczane w katalogach dostępnych w ścieżce systemowej, a więc mogą być uruchamiane bez żadnej dodatkowej konfiguracji zmiennej PATH, poleceń zmieniających katalog roboczy czy dodawania pełnej ścieżki do polecenia. Program `py` uruchamia interaktywną sesję Pythona w wersji konsolowej, a polecenie `pyw` uruchamia sesję z graficznym interfejsem użytkownika:

```
c:\code> py  
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit  
...  
Type "help", "copyright", "credits" or "license" for more information.  
>>> ^Z  
c:\code> py -2  
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] ...  
Type "help", "copyright", "credits" or "license" for more information.  
>>> ^Z
```

```
c:\code> py -3.1
Python 3.1.4 (default, Jun 12 2011, 14:16:16) [MSC v.1500 64 bit (AMD64)] ...
Type "help", "copyright", "credits" or "license" for more information.

>>> ^Z
```

Jak pokazano w dwóch ostatnich przykładach, argumentami wywołania launcherów z poziomu wiersza poleceń mogą również być numery wersji Pythona (można ich także używać w wierszach `#!` uniksowych skryptów, o czym będziemy mówić nieco później). Launchery pozwalają na uruchamianie plików zawierających kod programów w języku Python, tak jak to robiliśmy wcześniej z oryginalnym plikiem `python.exe`, który jest nadal dostępny i działa tak jak do tej pory (choć launchery niejako dublują w jakiś sposób część jego funkcjonalności).

Programy uruchamiające (launcher) są standardową częścią Pythona 3.3 i są dostępne jako samodzielny dodatek do innych wersji. W tym i późniejszych rozdziałach opowiem jeszcze więcej na temat launcherów, włącznie z krótkim omówieniem ich wsparcia dla wiersza `#!(shebang)` w skryptach. Ponieważ jednak jest to temat interesujący tylko dla użytkowników systemu Windows, a nawet i dla nich jest domyślnie ograniczony tylko do grupy używającej wersji 3.3, większość szczegółowych informacji dotyczących programów uruchamiających zebrałem w dodatku B.

Jeżeli masz zamiar pracować z Pythonem 3.3 lub nowszym na platformie Windows, proponuję, abyś zajrzał teraz do tego dodatku, ponieważ launcher zapewniają alternatywny, a pod pewnymi względami nawet lepszy sposób uruchamiania poleceń i skryptów Pythona. Na poziomie podstawowym w większości przykładów przedstawionych w tej książce użytkownicy programu uruchamiającego mogą po prostu zamiast polecenia `python` wpisywać `py`, a co więcej, mogą również uniknąć pewnych kroków konfiguracji. Zalety launcherów widać zwłaszcza na komputerach z wieloma wersjami Pythona; w takim przypadku nowy program uruchamiający daje znacznie lepszą kontrolę nad tym, który Python uruchamia Twój kod.

Gdzie zapisywać programy – katalogi z kodem źródłowym

Skoro już zaczeliśmy mówić o uruchamianiu programów, chciałbym powiedzieć kilka słów o tym, *gdzie* przechowywać pliki z kodem źródłowym. Aby zachować prostotę wywodu, zarówno w tym rozdziale, jak i w całej książce, będę uruchamiać programy z katalogu roboczego `C:\code`, który utworzyłem na moim komputerze z systemem Windows. To właśnie będzie miejsce, z którego będę rozpoczynać większość sesji interaktywnych i gdzie będę zapisywać i uruchamiać większość skryptów. Oznacza to również, że pliki tworzone w różnych przykładach, będą również w większości zapisywane w tym katalogu.

Jeżeli będziesz chciał pracować z tą książką w sposób w pełni interaktywny, przed rozpoczęciem powinieneś prawdopodobnie zrobić coś podobnego. Jeżeli potrzebujesz pomocy przy tworzeniu katalogu roboczego na swoim komputerze, to poniżej znajdziesz kilka wskazówek:

- W systemie Windows można utworzyć katalog dla kodów źródłowych za pomocą Eksploratora plików lub z poziomu wiersza poleceń. W Eksploratorze plików poszukaj przycisku *Nowy folder*, zajrzyj do menu *Plik* lub spróbuj kliknąć prawym przyciskiem myszy. W wierszu polecenia wpisz i uruchom komendę `mkdir`, zwykle po przejściu do żądanego katalogu nadrzędnego (np. `cd c:\`, a następnie `mkdir code`). Twój katalog roboczy może się znajdować w dowolnym miejscu i nazywać jak chcesz, i wcale nie musi to być `C:\code` (wybrałem tę nazwę, ponieważ jest krótka i dzięki temu nie zajmuje dużo miejsca w przykładach). Używanie jednego, centralnego katalogu do wszystkich ćwiczeń ułatwi Ci pracę i pomoże uprościć niektóre zadania. Więcej wskazówek oraz informacji na takie tematy znajdziesz w ramce „Gdzie w systemie Windows można znaleźć wiersz poleceń?” oraz w dodatku A.

- W systemach opartych na systemie Unix (w tym macOS i Linux) Twój katalog roboczy może znajdować się w katalogu domowym `/usr/home` i może zostać utworzony z poziomu wiersza poleceń powłoki systemu za pomocą polecenia `mkdir` lub eksploratora plików wyposażonego w graficzny interfejs użytkownika, ale wszędzie obowiązują te same zasady. Podobnie będzie, jeżeli w systemie Windows używasz pakietu Cygwin, udostępniającego w pełni funkcjonalną powłokę uniksową, chociaż w takiej sytuacji nazwy katalogów mogą się nieco różnić (`/home` i `/cygdrive/c` są tu dobrymi przykładami).

Kod źródłowy możesz przechowywać również w katalogu instalacyjnym Pythona (np. `C:\Python33` w systemie Windows), co pozwala uprościć niektóre wiersze polecień przed ustawieniem zmiennej `PATH`, ale raczej nie powinieneś tego robić — katalog ten przeznaczony jest dla samego Pythona, a Twoje pliki mogą zostać przypadkowo skasowane podczas instalowania kolejnej aktualizacji Pythona lub podczas odinstalowywania jego starej wersji.

Od tej chwili pracę z przykładami omawianymi w tej książce powinieneś zawsze zaczynać od przejścia do katalogu roboczego. Przykłady w książce, które pokazują katalog, będą odzwierciedlały katalog roboczy mojego laptopa z systemem Windows; kiedy widzisz `C:\code>` lub `%`, pomyśl o lokalizacji i nazwie własnego katalogu roboczego.

Czego nie wpisywać — znaki zachęty i komentarze

Skoro mówimy o znakach zachęty, pamiętaj, że w tej książce czasami pokazujemy systemowe znaki zachęty w postaci ogólnego symbolu `%`, a czasami w pełnym formacie `C:\code>`. Pierwszy z nich ma być niezależny od platformy (a wywodzi się z wcześniejszych wersji Linuksa), a drugi jest używany w kontekstach specyficznych dla systemu Windows. Po znaku zachęty dodaję także dodatkową spację, co ma na celu zwiększenie czytelności przykładów przedstawionych w tej książce. Innymi słowy, znak `%` na początku wiersza polecień reprezentuje znak zachęty Twojego systemu, jakkolwiek miałby on wyglądać. Na przykład na moim komputerze znak `%` w wierszu polecień systemu Windows oznacza znak zachęty `C:\code>`, a w mojej instalacji Cygwin reprezentuje znak zachęty `$`.

Uwaga dla mniej doświadczonych użytkowników: przepisując kod przykładów nie wpisuj znaków `%` (lub innych znaków zachęty, na przykład `C:\code>`), które widzisz na listingach — jest to tekst, który Twój system wyświetla na ekranie. Wpisuj tylko tekst znajdujący się po znaku zachęty. Podobnie, nie przepisuj znaków `>>>` i `...` wyświetlanych na początku wierszy w listingach obrazujących interakcję z interpreterem — są to znaki zachęty, które Python wyświetla automatycznie w celu ułatwienia interaktywnego wprowadzania kodu. Wpisuj tylko tekst znajdujący się po takich znakach zachęty. Na przykład znak `...` jest używany w niektórych powłokach do wskazywania wiersza kontynuacji, ale nie pojawia się w środowisku IDLE. Znajdziesz go również w niektórych listingach (ale nie wszystkich) zamieszczonych w tej książce; nie powinieneś go przepisywać nawet jeżeli Twoja powłoka go nie wyświetla.

Aby ułatwić Ci zapamiętanie, informacje wpisywane przez użytkownika są w przykładach wyróżnione pogrubioną czcionką, a komunikaty wyświetlane przez system są napisane zwykłą czcionką. W niektórych systemach znaki zachęty mogą być nieco inne (na przykład, skoncentrowana na wydajności dystrybucja PyPy, o której pisaliśmy w rozdziale 2., używa czteroznakowych ciągów `>>>` i `....`), ale zastosowanie mają te same zasady. Należy również pamiętać, że polecenia wpisywane po systemowych bądź „pythonowych” znakach zachęty mają być uruchamiane natychmiast i nie są zwykle zapisywane w plikach kodu źródłowego; niebawem zobaczysz, dlaczego to rozróżnienie jest takie ważne.

Z tego samego powodu zazwyczaj nie musisz wpisywać tekstu, który w listingach zaczyna się od znaku `#` — jak się niebawem przekonasz, są to wiersze komentarzy, a nie kod, który będzie wykonywany. W większości przypadków możesz bezpiecznie zignorować tekst zamieszczany po znaku `#` — jedynym wyjątkiem są sytuacje, w których znak `#` jest używany w pierwszych wierszach skryptu do wprowadzenia określonej dyrektywy dla systemu Unix lub programu uruchamiającego Pythona 3.3 w systemie Windows (więcej szczegółowych informacji na temat

Pythona w systemie Unix i programów uruchamiających znajdziesz w dalszej części tego rozdziału oraz w dodatku B).



Jeżeli pracując z książką zamierzasz samodzielnie wykonywać omawiane przykłady, zwróć uwagę, że począwszy od rozdziału 17., na wielu listingach przedstawiających interakcję z konsolą, trzy kropki (...) reprezentujące znaki kontynuacji wiersza są celowo pomijane, tak aby ułatwić Ci kopiowanie i wklejanie większych fragmentów kodu, takich jak całe funkcje czy klasy z elektronicznego wydania tej książki. We wcześniejszych przykładach powinieneś po prostu kopiować lub przepisywać po jednym wierszu i pomijać znaki kontynuacji wierszy i znaki zachęty powłoki. Takie podejście jest celowe. Na początku pracy z nowym językiem programowania ręczne wpisywanie kodu jest niezmiernie ważne i pozwala na poznanie szczegółów składni oraz błędów. Niektóre przykłady będą dostępne w postaci listingów lub plików dostępnych w pakietie przykładów książki (tak jak pisaliśmy we wstępie). W książce często przełączamy się między formatami listingów; w razie wątpliwości pamiętaj, że jeżeli zobaczyz znak zachęty >>>, będzie to oznaczało, że kod jest wpisywany interaktywnie.

Interaktywne wykonywanie kodu

Po tym nieco przydługim wstępnie możemy w końcu przejść do wpisywania jakiegoś rzeczywistego kodu. Bez względu na sposób uruchomienia sesja interaktywna Pythona rozpoczyna się od wyświetlenia na ekranie dwóch wierszy z informacjami o wersji Pythona i kilku poleceniach pomocy (w większości przykładów oba wiersze zostały usunięte w celu zaoszczędzenia miejsca), a następnie na ekranie pojawia się znak zachęty >>>, po którym możesz rozpocząć wpisywanie poleceń Pythona lub innych wyrażeń.

W interaktywnej sesji konsoli Pythona, wynik działania wpisanego polecenia wyświetlany jest w kolejnym wierszu natychmiast po naciśnięciu klawisza *Enter*. Na przykład, poniżej prezentujemy wyniki działania dwóch instrukcji `print` (w Pythonie 3.x `print` jest tak naprawdę wywołaniem funkcji, ale już w wersjach 2.x tak nie jest, stąd nawiasy wymagane są jedynie w 3.x):

```
% python
>>> print('Witaj, świecie!')
Witaj, świecie!
>>> print(2 ** 8)
256
```

A więc stało się — właśnie uruchomiłeś swój pierwszy kod Pythona (nie spodziewałeś się chyba tutaj zobaczyć hiszpańskiej inkwizycji?...). Nie musisz jeszcze się przejmować szczegółami działania instrukcji `print` (składnią zajmiemy się w kolejnym rozdziale). W skrócie, instrukcje wypisują odpowiednio ciąg znaków oraz liczbę będącą rezultatem obliczeń, co widać w wierszach wyników, które następują po każdym wierszu polecenia, rozpoczętym znakiem zachęty (>>>). Wyrażenie `2 ** 8` oznacza w języku Python podniesienie liczby 2 do potęgi 8.

Kiedy pracujemy w sposób interaktywny, możemy wpisać dowolną liczbę poleceń. Każde jest wykonywane natychmiast po wpisaniu. Co więcej, ponieważ sesja interaktywna automatycznie wyświetla wyniki wpisywanych wyrażeń, zazwyczaj nie trzeba nawet w jawnym sposobie nakazywać wyświetlenia ich za pomocą instrukcji `print`:

```
>>> drwal = 'OK'
```

```
>>> drwal
'OK'
>>> 2 ** 8
256
>>> ^Z      # Aby zakończyć sesję, naciśnij Ctrl+D (Unix) lub Ctrl+Z
(Windows)
%
```

W powyższym kodzie pierwszy wiersz zapisuje wartość, przypisując ją do *zmiennej* (`drwal`). Dwa ostatnie, wpisywane wiersze (`drwal` oraz `2 ** 8`) są *wyrażeniami*, a ich wyniki wyświetlane są automatycznie. Aby wyjść z sesji interaktywnej Pythona i powrócić do wiersza poleceń powłoki, należy na komputerach z systemem Unix użyć skrótu klawiaturowego `Ctrl+D`, natomiast w systemie Windows — `Ctrl+Z`. Jeżeli pracujesz w środowisku graficznym IDLE, powinieneś użyć skrótu `Ctrl+D`, albo po prostu zamknąć okno.

Zwróć uwagę na fragment tekstu zapisany pochyloną czcionką w przedostatnim wierszu naszego przykładowego listingu (który zaczyna się od znaku `#`). Takiego zapisu będę używał do oznaczania swoich uwag do omawianego listingu. Pamiętaj, że pracując w sesji interaktywnej powłoki lub konsoli Pythona nie powinieneś przepisywać takich komentarzy; tekst znajdujący się po znaku `#` jest co prawda traktowany przez Pythona jako komentarz, ale podczas pracy z wierszem poleceń powłoki systemowej może spowodować błąd.

W sesji interaktywnej Pythona nie zrobiliśmy do tej pory niczego wielkiego — ot, wpisaliśmy kilka instrukcji `print` i instrukcji przypisania, a także kilka wyrażeń, które bardziej szczegółowo omówimy później. Najważniejszą kwestią jest jednak to, że interpreter wykonuje kod wpisany w każdym wierszu natychmiast po naciśnięciu przycisku *Enter*.

Na przykład, kiedy po znaku zachęty `>>>` wpisaliśmy pierwszą instrukcję `print`, wyniki działania (łańcuch znaków Pythona) zwrócone zostały od razu. Nie trzeba było tworzyć pliku z kodem źródłowym ani uprzednio kompilować kodu, co byłoby konieczne w przypadku innych języków, takich jak C czy C++. Jak zobaczymy w kolejnych rozdziałach, w sesji interaktywnej można również wpisywać instrukcje kilkuwierszowe. Takie instrukcje wykonywane są po wpisaniu wszystkich wierszy i dwukrotnym naciśnięciu klawisza *Enter* w celu dodania pustego wiersza.

Do czego służy sesja interaktywna

Sesja interaktywna wykonuje kod i od razu zwraca wyniki działania kolejnych wpisywanych poleceń, jednak nie zapisuje ich w pliku. Choć w praktyce w sesji interaktywnej nie piszemy zazwyczaj większych programów, jest to świetna opcja do *eksperymentowania* z językiem i *testowania* programów.

Eksperymentowanie

Ponieważ kod jest tam wykonywany natychmiast, sesja interaktywna jest idealnym miejscem na eksperymenty z językiem programowania, stąd w tej książce będzie często wykorzystywana do demonstrowania krótszych przykładów. Tak naprawdę to pierwsza reguła, jaką należy zapamiętać: jeżeli masz jakiekolwiek wątpliwości co do tego, jak działa dany fragment kodu w Pythonie, wystarczy rozpoczęć sesję interaktywną i sprawdzić, co się stanie.

Załóżmy na przykład, że czytasz kod programu napisanego w Pythonie i natrafiasz na wyrażenie takie jak `'Mielonka!' * 8`, którego znaczenia nie rozumiesz. W tym momencie możesz poświęcić dziesięć minut na przedzieranie się przez dokumentację i książki, aby dowiedzieć się, co robi ten kod, lub zamiast tego możesz po prostu wykonać go w sesji interaktywnej:

```
% python  
>>> 'Mielonka!' * 8 # Nauka przez próbowanie  
'Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!'
```

Natychmiastowe informacje zwrotne otrzymywane w sesji interaktywnej to najczęściej najszybszy sposób na nauczenie się, co robi jakiś fragment kodu. Po wykonaniu takiego polecenia stanie się dla Ciebie jasne, że taki kod wyświetla powtórzenia łańcucha znaków. W Pythonie znak `*` dla liczb oznacza mnożenie, jednak w przypadku łańcuchów znaków — powtarzanie. Przypomina to wielokrotne łączenie tego samego ciągu znaków (więcej informacji o ciągach znaków znajdziesz w rozdziale 4.).

Istnieje całkiem spora szansa, że eksperymentując w ten sposób niczego nie zepsujesz — przynajmniej na razie. Aby wyrządzić jakieś prawdziwe szkody, na przykład usunąć pliki czy wykonać niebezpieczne polecenia powłoki, musiałbyś się naprawdę postarać i zimportować wybrane moduły w jawnym sposobie (musiałbyś też dowiedzieć się czegoś więcej na temat sposobu działania interfejsów systemowych). Zwykły kod w języku Python niemal zawsze można bezpiecznie wykonać.

Zobaczmy na przykład co się stanie, kiedy *popełnisz błąd* w interaktywnej sesji Pythona:

```
>>> X # Robimy błąd  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    NameError: name 'X' is not defined
```

W Pythonie użycie zmiennej przed przypisaniem do niej wartości zawsze jest błędem (w przeciwnym razie, gdyby zmienne były wypełniane za pomocą wartości domyślnych, pewne błędy mogłyby pozostać nieauważone). Oznacza to na przykład, że musisz zainicjować wartości liczników, zanim będziesz mógł je inkrementować, musisz zainicjować listy przed dodawaniem do nich nowych elementów itd. — nie deklarujesz zmiennych, ale zanim będziesz mógł ich używać, musisz do nich przypisać jakąś wartość.

Później dowieš się więcej na ten temat; ważne jest jednak to, że robiąc tego typu błędy, nie powodujesz zakończenia działania Pythona czy komputera. Zamiast tego otrzymasz komunikat o błędzie wskazujący miejsce pomyłki wraz z wierszem kodu i możesz kontynuować sesję interaktywną czy skrypt. Po zapoznaniu się z Pythonem komunikaty o błędach wyświetlane przez interpreter często w zupełności wystarczą do debugowania kodu (więcej informacji o debugowaniu znajduje się w ramce „Debugowanie kodu w Pythonie” na końcu rozdziału).

Testowanie

Interaktywny interpreter Pythona oprócz tego, że służy jako narzędzie do eksperymentowania podczas nauki języka, jest również idealnym miejscem na testowanie kodu zapisanego w plikach. Pliki modułów można zimportować interaktywnie; można również testować programy zapisane w takich plikach, wpisując odpowiednie wywołania z poziomu wiersza poleceń sesji interaktywnej.

Na przykład, poniższy kod testuje funkcję w gotowym module udostępnianym wraz z Pythonem w jego bibliotece standardowej (badana funkcja wyświetla nazwę katalogu, w którym aktualnie pracujesz, wstawiając w ścieżce podwójne znaki lewego ukośnika). To samo będziesz mógł zrobić dla własnych plików modułów, kiedy już zaczniesz je tworzyć:

```
>>> import os  
>>> os.getcwd() # Testowanie w locie  
'c:\\\\code'
```

Mówiąc bardziej ogólnie, w sesji interaktywnej można testować komponenty programu bez względu na ich źródło — można w niej importować i testować funkcje oraz klasy z własnych plików programów napisanych w Pythonie, wpisać wywołania do połączonych funkcji języka C czy korzystać z klas Javy pod Jythonem. Po części ze wzgledu na interaktywną naturę Python obsługuje eksperymentalny i eksploracyjny styl programowania, który przyda się na początku przygody z tym językiem. Chociaż bardziej doświadczeni programiści Pythona często testują również kod zapisany w pliku (a nauczysz się to robić w prosty sposób w dalszej części książki), dla wielu użytkowników interaktywna sesja konsoli Pythona jest nadal ich pierwszą linią testowania programu.

Uwagi praktyczne – wykorzystywane sesji interaktywnej

Chociaż korzystanie z interaktywnej sesji Pythona jest łatwe, warto przytoczyć kilka wskazówek, o których powinni pamiętać zwłaszcza poczatkujący użytkownicy tego języka. Zarówno poniżej, jak i w kilku innych miejscach tego rozdziału zamieszczamy szereg wskazówek, których wykorzystanie może zaoszczędzić Ci kłopotów w przyszłości:

- **Wpisuj jedynie polecenia Pythona.** Przede wszystkim pamiętaj o tym, że w interaktywnej sesji Pythona powinieneś wpisywać jedynie polecenia Pythona, a nie wszelkie inne polecenia systemowe. Oczywiście istnieją sposoby na wykonywanie poleceń systemowych z poziomu kodu Pythona (na przykład za pomocą funkcji `os.system`), ale nie są one tak proste i bezpośrednie, jak wpisywanie poleceń od razu w wierszu poleceń konsoli Pythona.
- **Instrukcje print wymagane są jedynie w plikach.** Ponieważ interaktywny interpreter Pythona automatycznie wyświetla wyniki wyrażeń, wpisywanie instrukcji `print` w sesji interaktywnej nie jest konieczne. Jest to miła opcja, ale często myląca dla użytkowników, którzy przechodzą do zapisywania kodu w plikach — w programie zapisanym w pliku, żeby wyniki działania zostały wyświetlane na ekranie, niezbędne jest użycie instrukcji `print`, ponieważ wyniki działania wyrażeń nie są zwracane automatycznie. Trzeba pamiętać, że instrukcja `print` jest niezbędna w plikach, natomiast w sesji interaktywnej jej użycie jest opcjonalne.
- **Nie stosuj (na razie) wcięć w kodzie wpisywanym w sesji interaktywnej.** Kiedy piszesz programy w Pythonie, obojętnie, czy interaktywnie, czy w pliku, powinieneś pamiętać o rozpoczęaniu wszystkich niezagieźdzonych instrukcji w pierwszej kolumnie (to znaczy jak najbardziej po lewej stronie). Jeżeli tak się nie zrobisz, Python może zwrócić komunikat o błędzie składni (`SyntaxError`), ponieważ odstęp po lewej stronie kodu uznawane są za wcięcia grupujące instrukcje zagnieżdżone. Aż do rozdziału 10. wszystkie pisane przez nas instrukcje będą niezagieździone, dlatego taka informacja powinna Ci na razie wystarczyć. Przy pierwszych eksperymentach z Pythonem kwestia ta nieodmiennie wprowadza użytkowników w zakłopotanie. Pamiętaj, że wiodące spacje mogą generować komunikat o błędzie, więc pracując w interaktywnej sesji Pythona nie zaczynaj wiersza kodu od spacji lub tabulatora, chyba że jest to kod zagnieżdżony.
- **Uważaj na zmianę znaku zachęty w instrukcjach złożonych.** Instrukcje `złożone` (wielowierszowe) pojawią się w rozdziale 4., a na bardziej poważnie dopiero w rozdziale 10., jednak warto wiedzieć, że kiedy w sesji interaktywnej wpisujesz kolejny wiersz instrukcji złożonej, znak zachęty może się zmienić. W oknie powłoki Pythona znak zachęty zmienia się z `>>>` na `...` dla drugiego i każdego kolejnego wiersza. W środowisku graficznym IDLE kolejne wiersze instrukcji złożonych są wcinane automatycznie.

W rozdziale 10. przekonasz się, dlaczego jest to takie ważne. Na razie, jeżeli w czasie wpisywania kodu napotkasz znak zachęty w postaci trzech kropiek lub pustego wiersza, będzie to oznaczało najprawdopodobniej, że w jakiś sposób zmyliłeś Pythona, który sądzi, że wpisujesz instrukcję wielowierszową. Aby powrócić do normalnego znaku zachęty, powinieneś nacisnąć klawisz `Enter` lub wybrać z klawiatury skrót `Ctrl+C`. Znaki zachęty

>>> oraz ... można zmienić (są one dostępne we wbudowanym module sys), jednak w przykładach zamieszczonych w książce zakładam, że taka zmiana nie została wykonana.

- **Instrukcje złożone w sesji interaktywnej powinieneś kończyć pustym wierszem.** W sesji interaktywnej wstawienie pustego wiersza (naciśnięcie klawisza *Enter* na początku wiersza) jest niezbędne do poinformowania interaktywnego Pythona, że skończyłeś już wpisywać instrukcję wielowierszową. Oznacza to zatem, że w celu wykonania instrukcji złożonej powinieneś dwukrotnie nacisnąć klawisz *Enter*. W kodzie zapisanym w plikach puste wiersze nie są z kolei wymagane, a jeżeli są obecne — zostaną zignorowane. Jeżeli w sesji interaktywnej na końcu instrukcji złożonej nie naciśniesz klawisza *Enter* dwukrotnie, będzie Ci się wydawać, że pozostajesz w stanie zawieszenia, ponieważ interpreter interaktywny nic nie zrobi — po prostu będzie czekać, aż znowu naciśniesz klawisz *Enter*!
- **Sesja interaktywna wykonuje po jednym wierszu naraz.** W sesji interaktywnej kolejne polecenie możesz wykonać dopiero po zakończeniu działania poprzedniego polecenia. W przypadku prostych instrukcji jest to całkiem naturalne, ponieważ naciśnięcie klawisza *Enter* wykonuje wpisaną instrukcję. W przypadku instrukcji złożonych powinieneś pamiętać, aby zakończyć ją wstawieniem pustego wiersza i wykonaniem jej przed rozpoczęciem wpisywania kolejnego polecenia.

Wpisywanie instrukcji wielowierszowych

Choć ryzykuję, że będę się powtarzał, w trakcie pracy nad uaktualnieniem tego rozdziału otrzymałem wiele emaili od Czytelników, którzy natrafili na jakieś niespodzianki w związku z dwoma ostatnimi punktami — dlatego kwestia ta zasługuje na dodatkowe wyjaśnienia. Instrukcje wielowierszowe (inaczej: złożone) wprowadzone zostaną w kolejnym rozdziale, a ich składnią zajmiemy się bardziej formalnie w dalszej części książki. Ponieważ jednak ich zachowanie różni się nieco w plikach i w sesji interaktywnej, warto zwrócić szczególną uwagę na dwie kwestie.

Po pierwsze, w sesji interaktywnej powinieneś pamiętać o zakończeniu wielowierszowych instrukcji złożonych (takich jak pętle *for* oraz testy *if*) pustym wierszem. Innymi słowy, aby zakończyć i wykonać instrukcję wielowierszową, *klawisz Enter powinieneś nacisnąć dwa razy*. Przykładowo:

```
>>> for x in 'mielonka':  
...     print(x)          # Dwukrotne naciśnięcie Enter w celu wykonania tej  
pętli  
...  
...
```

W pliku skrytu pusty wiersz po instrukcji złożonej nie jest jednak potrzebny — jest to wymagane *jedynie* w sesji interaktywnej. W plikach puste wiersze nie są wymagane, a jeżeli są obecne — zostaną zignorowane. W sesji interaktywnej służą one do zakończenia instrukcji wielowierszowych. Przypomnienie: znak kontynuacji wiersza *...* jest automatycznie wyświetlany w interaktywnej konsoli Pythona jako wskaźnik wizualny; może nie pojawiać się w Twoim środowisku (jeżeli używasz np. IDLE) i czasami jest też celowo pomijany w listingach zamieszczonych w tej książce, ale nie wpisuj go samodzielnie, jeżeli go nie ma.

Pamiętaj również, że sesja interaktywna wykonuje *po jednej instrukcji naraz*. W celu wykonania pętli lub innej instrukcji wielowierszowej przed wpisaniem kolejnej instrukcji powinieneś dwukrotnie nacisnąć klawisz *Enter*.

```
>>> for x in 'mielonka':  
...     print(x)          # Przed wpisaniem nowej instrukcji dwukrotnie  
naciśnij klawisz Enter  
...     print('gotowe')
```

```
File "<stdin>", line 3  
    print('gotowe')  
    ^  
  
SyntaxError: invalid syntax
```

Oznacza to, że nie można kopiować i wklejać większej liczby wierszy kodu do sesji interaktywnej, o ile kod nie zawiera pustych wierszy po każdej instrukcji złożonej. Taki kod lepiej jest wykonać w pliku — co jest tematem kolejnego podrozdziału.

Systemowy wiersz poleceń i pliki źródłowe

Choć sesja interaktywna doskonale nadaje się do eksperymentów i testów, ma jedną ogromną wadę: programy napisane w konsoli Pythona znikają od razu po wykonaniu przez interpreter. Kod wpisywany interaktywnie nie jest zapisywany w pliku, dlatego nie można go uruchomić ponownie bez wpisywania go kolejny raz. Co prawda skopiowanie polecenia i ponowne wklejenie go może tu nieco pomóc, ale nie na dużą skalę, zwłaszcza kiedy zaczniesz pisać większe programy. Aby skopiować kod z sesji interaktywnej i wkleić go ponownie, trzeba by z niego wyciąć na przykład znaki zachęty Pythona czy wyniki działania polecień — nie jest to do końca zgodne z metodologią nowoczesnego programowania!

Aby zapisać program na stałe, musisz wpisać kod do plików, które są zazwyczaj nazywane *modułami*. Moduły to po prostu pliki tekstowe zawierające instrukcje Pythona. Po ich utworzeniu można nakazać interpreterowi Pythona wykonanie tych instrukcji w każdym pliku dowolną liczbę razy i na różne sposoby — za pomocą wiersza poleceń, kliknięcia ikony pliku czy z wykorzystaniem odpowiedniego polecenia środowiska IDLE. Bez względu na sposób uruchomienia Python wykonuje kod zapisany w module po kolei z góry do dołu.

Terminologia używana w tej dziedzinie może być różna. Na przykład, pliki modułów języka Python nazywane są czasami *programami* — programem jest zatem nazywana seria zakodowanych wcześniej instrukcji przechowywanych w plikach, które można wielokrotnie wykonywać. Pliki modułów wykonywane w sposób bezpośredni są czasami nazywane również *skryptami* — co jest nieformalnym terminem oznaczającym zazwyczaj plik programu najwyższego poziomu. Niektóre rezerwują termin „moduł” dla pliku zaimportowanego z innego pliku i „skrypt” dla głównego pliku programu — i takiej terminologii będziemy się również trzymać w tej książce (w dalszej części tego rozdziału dowiesz się bardziej dokładnie, czym są pliki główne, jak się importuje moduły i czym jest program „najwyższego poziomu”).

Bez względu na nazewnictwo na kolejnych kilku stronach omówimy sposoby wykonywania kodu zapisanego w plikach modułów. Na początek nauczymy się, jak można w najprostszy sposób uruchomić taki plik bezpośrednio z poziomu wiersza poleceń powłoki systemowej, podając jego nazwę jako argument wywołania polecenia `python`. Chociaż niektórym użytkownikom może się to wydawać prymitywne — i często można tego uniknąć, używając środowiska graficznego takiego jak IDLE, o którym opowiem nieco później — dla wielu doświadczonych programistów okno wiersza poleceń powłoki systemowej, wraz z oknem edytora tekstu, to jedyne „zintegrowane środowisko programistyczne”, jakiego będą kiedykolwiek potrzebować, zapewniające im bardziej bezpośrednią kontrolę nad tworzonymi programami.

Pierwszy skrypt

A zatem zaczynamy. Uruchom swój ulubiony edytor tekstu (na przykład *vi*, Notatnik czy edytor środowiska IDLE), a następnie w katalogu przeznaczonym na kod źródłowy utwórz nowy plik tekstowy o nazwie *script1.py* i wpisz do niego poniższą sekwencję poleceń:

```
# Pierwszy skrypt w Pythonie
import sys                                # Załadowanie modułu biblioteki
print(sys.platform)
print(2 ** 100)                            # Podniesienie 2 do potęgi
x = 'Mielonka!'
print(x * 8)                             # Powtórzenie łańcucha znaków
```

Jest to nasz pierwszy oficjalny skrypt napisany w Pythonie (nie licząc dwuwierszowego mini-skryptu z rozdziału 2.). Nie powinieneś się jeszcze szczególnie przejmować utworzonym kodem; w skrócie program ten:

- Importuje moduł Pythona (bibliotekę dodatkowych narzędzi), za pomocą którego pobiera nazwę platformy.
- Wykonuje trzy wywołania funkcji `print`, które wyświetlają wyniki działania skryptu.
- Wykorzystuje zmienną `x` utworzoną w momencie przypisania do ciągu znaków.
- Wykonuje różne działania na obiektach, które zaczniemy omawiać w kolejnym rozdziale.

Parametr `sys.platform` to po prostu ciąg znaków identyfikujący typ komputera, na jakim pracujemy. Informacje takie znajdują się w standardowym module o nazwie `sys`, który należy zaimportować do skryptu (więcej na temat importowania powiemy już za chwilę).

Aby nieco urozmaicić kod naszego skryptu, dodałem do niego również kilka *komentarzy* Pythona, czyli tekst znajdujący się po znaku `#`. Wspomniałem o nich wcześniej, ale teraz uczynię to w sposób bardziej formalny, ponieważ zaczynamy ich używać w naszych skryptach. Komentarze mogą się pojawiać w osobnych wierszach lub w tym samym wierszu po prawej stronie kodu. Tekst po znaku `#` jest ignorowany i traktowany jako komentarz przeznaczony tylko dla ludzi; nie jest on uznawany za część składni poleceń. Przy kopiowaniu kodu możesz swobodnie zignorować wszystkie komentarze — spełniają one wyłącznie rolę informacyjną. W książce zazwyczaj wykorzystujemy inny styl formatowania do wizualnego wyróżnienia komentarzy, jednak w kodzie pojawiają się one jako zwykły tekst.

Nie musisz jeszcze teraz skupiać się na składni kodu znajdującego się w pliku, wszystkiego dowiesz się później. Zauważ jednak, że powyższy kod został zapisany w pliku, a nie w sesji interaktywnej, czyli udało nam się stworzyć w pełni funkcjonalny skrypt w języku Python.

Zwróc uwagę, że nasz plik modułu nosi nazwę *script1.py*. Tak jak wszystkie inne pliki najwyższego poziomu, mógłby się nazywać po prostu *script1*, jednak pliki zawierające kod, który chcemy *zaimportować* do innego programu, muszą mieć rozszerzenie *.py*. Importowanie zostanie omówione w dalszej części rozdziału. Ponieważ możesz zechcieć w przyszłości zaimportować taki plik do innych programów, dobrym rozwiązaniem będzie stosowanie rozszerzenia *.py* dla wszystkich plików zawierających kod źródłowy napisany w języku Python. Niektóre edytory tekstu wykrywają pliki Pythona po ich charakterystycznym rozszerzeniu. Jeżeli tego rozszerzenia nie będzie, być może niektóre opcje — jak kolorowanie elementów składni czy automatyczne wcięcia kodu — nie będą dostępne.

Wykonywanie plików z poziomu wiersza poleceń powłoki

Po zapisaniu pliku tekstowego zawierającego kod źródłowy, możesz poprosić Pythona, aby uruchomił go, podając pełną nazwę pliku jako pierwszy argument polecenia `python`,

wywoływanego z poziomu *wiersza poleceń powłoki systemu*, tak jak to zostało pokazane w przykładzie poniżej (nie wpisuj tego polecenia w interaktywnej sesji Pythona, a jeżeli samo polecenie nie zadziała od razu, przeczytaj następny akapit):

```
% python script1.py  
win32  
1267650600228229401496703205376  
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Możesz wpisać takie polecenie powłoki systemowej w dowolnym miejscu w systemie, w którym znajduje się wiersz poleceń — w oknie programu *Wiersz polecenia* systemu Windows, w oknie terminala *xterm* czy podobnym. Sprawdź, również, czy uruchamiasz program w tym samym katalogu roboczym, w którym zapisałś swój plik skryptu (w razie czego przejdź najpierw do odpowiedniego katalogu), i upewnij się, że uruchamiasz go w wierszu poleceń powłoki, a nie w interaktywnej sesji konsoli *>>>* Pythona. Jeżeli nie skonfigurowałeś wcześniej zmiennej średowiskowej PATH, pamiętaj, aby uzupełnić polecenie *python* pełną ścieżką do katalogu, tak jak to robiliśmy już wcześniej (chociaż nie jest to wymagane dla programu uruchamiającego *py* w systemie Windows oraz może nie być wymagane dla Pythona 3.3 i nowszych wersji).

Kolejna uwaga dla początkujących użytkowników: nie wpisuj pokazanego przed chwilą polecenia w pliku źródłowym *script1.py* utworzonym w poprzedniej sekcji. Przedstawione polecenie jest poleceniem systemowym, a nie kodem programu. Pierwszy wiersz listingu to polecenie powłoki używane do uruchomienia pliku źródłowego, a następujące po nim wiersze to wyniki działania wygenerowane przez instrukcje *print* w pliku kodu źródłowego. Przypominam, że znak % reprezentuje znak zachęty powłoki systemu i nie powinieneś go wpisywać samemu (nie marudź, to naprawdę bardzo często spotykany błąd!).

Jeżeli wszystko zadziała zgodnie z planem, powyższe polecenie powłoki sprawi, że Python wykona kod zapisany w pliku wiersz po wierszu, a na ekranie zobaczysz wyniki działania trzech instrukcji *print* ze skryptu — nazwę platformy (w postaci, w jakiej „widzi” ją Python), wartość będącą rezultatem podniesienia liczby 2 do potęgi 100 oraz rezultat działania wyrażenia powtarzającego łańcuch znaków, które pokazywaliśmy już wcześniej (więcej szczegółowych informacji na temat dwóch ostatnich polecień znajdziesz w rozdziale 4.).

Jeżeli jednak coś pójdzie nie tak, na ekranie pojawi się komunikat o błędzie. Powinieneś się wtedy upewnić, że kod został wprowadzony do pliku dokładnie tak, jak pokazano na listingu, a następnie spróbować ponownie. Opcje debugowania omówimy w ramce „Debugowanie kodu w Pythonie” na końcu rozdziału, a póki co najlepszym rozwiązaniem będzie dokładne skopiowanie kodu źródłowego prezentowanego na listingu. Jeżeli wszystko inne zawiedzie, możesz również spróbować uruchomić omawiane wcześniej środowisko graficzne IDLE — jest to narzędzie, które ukrywa przed użytkownikiem niektóre szczególności uruchamiania programów, choć czasami odbywa się to kosztem lepszej kontroli nad tym procesem, którą masz podczas pracy z poziomu wiersza poleceń.

Jeżeli przepisywanie polecień staje się dla Ciebie zbyt uciążliwe lub popełniasz przy tym zbyt wiele błędów, możesz także pobrać gotowe kody przykładów z serwera FTP wydawnictwa Helion, pod adresem <ftp://ftp.helion.pl/przyklady/pytho5.zip>, choć ręczne wpisywanie kodu z pewnością pomoże Ci nauczyć się unikania błędów składniowych. Szczegółowe informacje na temat sposobu pobierania plików przykładów dla tej książki możesz znaleźć w przedmowie.

Sposoby użycia wiersza poleceń

Ponieważ taki schemat wykorzystuje do uruchamiania programów napisanych w Pythonie wiersz polecenia powłoki systemu, zastosowanie mają wszelkie reguły składni powłoki. Można na przykład za pomocą specjalnej składni powłoki przekierować wyniki działania skryptu Pythona do pliku na dysku, dzięki czemu będziesz mógł do nich wrócić w przyszłości:

```
% python script1.py > saveit.txt
```

W tym przypadku trzy pokazane wcześniej wiersze wyników zostaną zapisane w pliku *saveit.txt* i nie będą wyświetlane w oknie wiersza polecenia. Takie działanie znane jest pod nazwą *przekierowania strumienia* (ang. *stream redirection*). Mechanizm ten jest dostępny dla danych na wejściu i wyjściu programów w systemach uniksowych oraz w systemie Windows. Jest to bardzo użyteczne do testowania kodu, ponieważ pozwala na pisanie programów, które mogą analizować zmiany w wynikach działania innych programów. Oczywiście nie ma to zbyt wiele wspólnego z samym Pythonem (który po prostu obsługuje ten mechanizm), dlatego pominimy tutaj szczegóły samej składni przekierowania.

Jeżeli korzystasz z platformy Windows, poniższy przykład zadziała w taki sam sposób, jednak systemowy znak zachęty jest zazwyczaj inny (o czym wspominaliśmy już wcześniej):

```
C:\code> python script1.py  
win32  
1267650600228229401496703205376  
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Jak poprzednio, jeżeli zmienna środowiskowa PATH nie została odpowiednio ustawiona, powinieneś podać pełną ścieżkę do Pythona lub wcześniej przejść do odpowiedniej lokalizacji za pomocą polecenia zmieniającego katalog roboczy:

```
C:\code> C:\python33\python script1.py  
win32  
1267650600228229401496703205376  
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Zamiast tego, jeżeli w systemie Windows używasz programu uruchamiającego, który pojawił się w Pythonie 3.3 (mówiliśmy o nim już wcześniej), możesz wykonać polecenia py, co da taki sam efekt, ale nie będzie wymagało podania ścieżki katalogu ani zmiany ustawień zmiennej środowiskowej PATH, a dodatkowo py pozwoli Ci wybrać numer wersji Pythona do uruchomienia skryptu:

```
c:\code> py -3 script1.py  
win32  
1267650600228229401496703205376  
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

We wszystkich wersjach systemu Windows można również wpisać nazwę skryptu i pominąć polecenie python. Ponieważ nowsze systemy Windows używają rejestru systemu Windows (mechanizmu skojarzenia nazw plików), aby znaleźć program, w którym można uruchomić dany plik, nie trzeba podawać nazwy polecenia python lub py w wierszu poleceń, aby uruchomić plik z rozszerzeniem .py. Poprzednie polecenie można zatem na większości komputerów z systemem Windows skrócić tak, jak to zostało pokazane w przykładzie poniżej — będzie ono automatycznie uruchamiane przez program python (dla wersji wcześniejszych niż 3.3) lub przez py w wersjach 3.3 i nowszych; działa to tak, jakbyś kliknął ikonę pliku .py w Eksploratorze (więcej na temat tej opcji powiemy już za chwilę):

```
C:\code> script1.py
```

Wreszcie należy pamiętać, by podać pełną ścieżkę do skryptu, jeżeli znajduje się on w katalogu innym niż ten, w którym aktualnie pracujesz. Poniższy wiersz polecenia uruchomiony w

katalogu `D:\other` zakłada, że Python podany jest w zmiennej środowiskowej PATH i uruchamia plik znajdujący się w innym katalogu:

```
C:\code> cd D:\other  
D:\other> python c:\code\script1.py
```

Jeżeli zmienna środowiskowa PATH nie zawiera katalogu Pythona, nie używasz programu uruchomieniowego py, a Python, ani plik skryptu nie znajdują się w katalogu, w którym aktualnie pracujesz, powinieneś dla obu plików użyć pełnych ścieżek:

```
D:\other> C:\Python33\python c:\code\script1.py
```

Uwagi praktyczne – wykorzystywanie wierszy poleceń i plików

Wykonywanie plików programów z poziomu systemowego wiersza poleceń jest relatywnie proste, w szczególności dla osób zaznajomionych z działaniem wiersza poleceń. Osoby początkujące powinny jednak zwrócić uwagę na kilka pułapek, których wyeliminowanie pozwoli uniknąć niepotrzebnych frustracji:

- **W systemie Windows i w środowisku IDLE powinieneś uważać na automatyczne rozszerzenia nazw plików.** Jeżeli pracujesz w systemie Windows i do pisania programów używasz Notatnika, przy zapisywaniu plików powinieneś pamiętać o wybraniu opcji *Wszystkie pliki* i nadaniu plikowi odpowiedniego rozszerzenia — .py. W przeciwnym razie Notatnik zapisze plik z domylnym rozszerzeniem .txt (na przykład jako *script1.py.txt*), co sprawi, że w niektórych sytuacjach nie będzie go można uruchomić albo zimportować.

Co gorsza, system Windows domyślnie ukrywa rozszerzenia plików, więc jeżeli nie zmieniłeś opcji ich wyświetlania, możesz nawet nie zauważyc, że kod zapisany został jako plik tekstowy, a nie plik Pythona. Może Ci to zdradzić ikona pliku — jeżeli nie widać w niej małego węża, możesz znaleźć się w opałach. Innymi symptomami tego problemu są niepokolorowany kod w edytorze IDLE i pliki, które zamiast się uruchomić, zostają otwarte do edycji.

Podobny problem występuje w programie Microsoft Word, który domyślnie dodaje on rozszerzenie .doc. Co gorsza jednak, dodaje również znaki formatowania niezgodne ze składnią Pythona. Generalnie przy zapisywaniu pliku w systemie Windows zawsze należy wybierać opcję *Wszystkie pliki* lub po prostu korzystać z przyjaznych programistom edytorów tekstowych, takich jak edytor środowiska IDLE. Co ciekawe, IDLE domyślnie nie dodaje nawet rozszerzenia .py, co podoba się wielu programistom, ale użytkownikom już nieco mniej.

- **Rozszerzenia plików oraz ścieżki do katalogów dodawaj przy uruchamianiu pliku z wiersza poleceń, ale nie używaj ich podczas importowania plików.** W systemowym wierszu poleceń nie można zapominać o podaniu pełnej nazwy pliku, czyli powinieneś korzystać z formy `python script1.py`, a nie `python script1`. Pamiętaj jednak, że instrukcja `import`, która zostanie omówiona w dalszej części rozdziału, pomija zarówno rozszerzenie .py, jak i ścieżkę do katalogu (czyli polecenia importowania plików mają formę `import script1`). Może się to wydawać trywialne, ale mylenie tych dwóch sytuacji jest częstym źródłem błędów.

W wierszu poleceń systemu pracujemy z powłoką systemu, a nie Pythona, dlatego pythonowe reguły wyszukiwania plików modułów nie mają tu zastosowania. Z tego powodu zawsze trzeba uwzględnić zarówno rozszerzenie .py, jak i pełną ścieżkę do katalogu, w którym znajduje się wykonywany plik (jeżeli to jest konieczne). Na przykład, aby wykonać plik znajdujący się w katalogu innym niż ten, w którym obecnie pracujesz, zazwyczaj powinieneś podać jego pełną ścieżkę (czyli `python d:\tests\spam.py`).

Wewnątrz kodu Pythona wystarczy jednak napisać `import spam`, a za odnalezienie pliku odpowiedzialny będzie mechanizm wyszukiwania modułów Pythona, który zostanie omówiony w dalszej części książki.

- **W plikach kodu powinieneś korzystać z instrukcji `print`.** Tak, mówiliśmy już o tym, ale jest to tak często popełniany błąd, że warto to powtórzyć jeszcze przynajmniej raz. W przeciwieństwie do kodu wpisywanego interaktywnie, by zobaczyć wyniki działania plików programów, powinieneś używać instrukcji `print`. Jeżeli na ekranie nie są wyświetlane żadne wyniki działania, powinieneś sprawdzić, czy w plikach kodu użyłeś polecenia `print`. Jeszcze raz przypominam, że instrukcje `print` *nie* są wymagane w sesji interaktywnej, ponieważ Python automatycznie wyświetla wyniki działania poleceń i wartości wyrażeń. Użycie `print` w takim kontekście nie jest błędem, ale wymaga niepotrzebnego pisania dodatkowych polecen.

Skrypty wykonywalne w stylu uniksowym — #!

Następny sposób uruchamiania skryptów jest w rzeczywistości wyspecjalizowaną formą poprzedniej techniki i pomimo tytułu tej sekcji, może on mieć zastosowanie do plików programów uruchamianych zarówno w systemach Unix, jak i Windows. Ponieważ jednak sposób ten ma swoje korzenie w systemie Unix, to właśnie tam rozpoczęmy naszą opowieść.

Podstawy skryptów uniksowych

Osoby korzystające z Pythona w systemach Unix, Linux czy innych systemach bazujących na Uniksie, mogą w prosty sposób zamienić pliki z kodem Pythona w programy wykonywalne, podobnie jak to się robi w przypadku skryptów pisanych w języku powłoki, takiej jak `csh` czy `ksh`. Takie pliki zazwyczaj nazywane są *skryptami wykonywalnymi* (ang. *executable scripts*). Uniksowe pliki wykonywalne to w uproszczeniu normalne pliki tekstowe zawierające instrukcje Pythona i mające dwie szczególne cechy charakterystyczne:

- **Ich pierwszy wiersz jest wierszem specjalnym.** Skrypty zazwyczaj rozpoczynają się od wiersza, w którym na początku znajdują się znaki `#!` (nazywa się je często *hash bang* lub *shebang*), a później ścieżka do interpretera Pythona zainstalowanego na komputerze.
- **Zazwyczaj mają prawa do wykonywania.** Pliki skryptów najczęściej oznaczone są jako wykonywalne, co mówi systemowi operacyjnemu, że mogą być wykonywane jako programy najwyższego poziomu. W systemach uniksowych zazwyczaj służy do tego polecenie `chmod +x plik.py`.

Przyjrzyjmy się teraz przykładowi przeznaczonemu dla systemu uniksowego. W edytorze tekstu utwórz plik o nazwie `brian` i wpisz do niego następujący kod:

```
#!/usr/local/bin/python  
print('Zawsze patrz ' + 'na jasną stronę życia')      # + oznacza w łańcuchach  
znaków konkatenację
```

Ten specjalny pierwszy wiersz mówi systemowi operacyjnemu, gdzie znajduje się interpreter Pythona. Z technicznego punktu widzenia taki wiersz jest komentarzem. Jak wspomniano wcześniej, wszystkie komentarze w programach w Pythonie rozpoczynają się od znaku `#` i rozciągają aż do końca wiersza. Są miejscem, w którym można w kodzie umieścić dodatkowe informacje przeznaczone dla innych osób. Kiedy jednak w pierwszym wierszu pojawia się komentarz pokazany powyżej, jest on traktowany w specjalny sposób, ponieważ system

operacyjny wykorzystuje go do odnalezienia interpretera, który może wykonać kod znajdujący się w dalszej części pliku.

Warto również zwrócić uwagę na to, że plik ten nazwany jest po prostu *brian*, bez rozszerzenia *.py* wykorzystywanego wcześniej w przypadku modułów. Dodanie *.py* do nazwy nie zaszkodzi (i może pomóc we wskazaniu, że ten plik jest programem Pythona), jednak ponieważ nie planujemy, by w przyszłości inne moduły importowały kod z tego pliku, jego nazwa jest bez znaczenia. Jeżeli za pomocą polecenia `chmod +x brian` nadamy temu plikowi prawa do wykonania, będziemy mogli uruchamiać go z poziomu powłoki systemu operacyjnego tak samo, jakby był programem binarnym (aby było to możliwe, upewnij się, że katalog `..`, czyli bieżący katalog roboczy, został dodany do zmiennej środowiskowej PATH; w przeciwnym wypadku musisz wywołać ten program w następujący sposób: `./brian`):

```
% brian
```

Zawsze patrz na jasną stronę życia

Sztuczka z wyszukiwaniem programu przy użyciu polecenia env w systemie Unix

W niektórych systemach uniksowych możemy uniknąć konieczności wpisywania ścieżki do interpretera Pythona na stałe poprzez umieszczenie w pierwszym wierszu komentarza *shebang* w następującej formie:

```
#!/usr/bin/env python  
...miejsce na skrypt...
```

Kiedy zapiszemy wiersz *shebang* w takiej postaci, polecenie *env* lokalizuje interpreter Pythona zgodnie z ustawieniami ścieżki wyszukiwania (w większości powłok uniksowych oznacza to przeszukanie wszystkich katalogów podanych w zmiennej środowiskowej PATH). Taki sposób postępowania powoduje, że skrypt jest łatwiejszy do przenoszenia na inne maszyny, gdyż w wierszu *shebang* nie musisz podawać pełnej ścieżki instalacyjnej Pythona (który na różnych maszynach może być zainstalowany w różnych miejscach).

Zakładając, że na każdej z maszyn masz dostęp do polecenia *env*, Twoje skrypty będą działać wszędzie, bez względu na miejsce zainstalowania Pythona w danym systemie. W praktyce zastosowanie zapisu z poleceniem *env* jest obecnie zalecane nawet dla tak popularnego programu jak `/usr/bin/python`, ponieważ na niektórych platformach Python może być zainstalowany w innym miejscu. Oczywiście takie rozwiązanie zakłada, że polecenie *env* znajduje się zawsze w tym miejscu (choć na niektórych systemach może znajdować się w katalogu `/sbin`, `/bin` lub gdzie indziej) — jeżeli tak nie jest, to wszystkie zalety tego rozwiązania stają się jego wadami!

Python 3.3 launcher — #! w systemie Windows

Informacja dla użytkowników systemu Windows korzystających z Pythona w wersji 3.2 i starszych — opisana tutaj metoda jest sztuczką zapożyczoną z systemu Unix i może nie działać na Waszych platformach. Nie ma się jednak czym martwić, w razie czego zawsze możecie skorzystać ze zwykłej techniki uruchamiania z wiersza poleceń, opisanej wcześniej. Nazwę pliku trzeba wtedy podać jako argument wywołania polecenia `python`:^[1]

```
C:\code> python brian
```

Zawsze patrz na jasną stronę życia

W takim przypadku nie potrzeba umieszczać specjalnego komentarza `#!` w pierwszym wierszu skryptu (choćż jeżeli będzie on obecny, Python po prostu go zignoruje), a sam plik kodu nie musi mieć uprawnień do wykonywania. W rzeczywistości, jeżeli chcesz przenosić pliki programów pomiędzy Unixem a Microsoft Windows i uruchamiać je w obu tych systemach, Twoje życie prawdopodobnie stanie się prostsze, jeżeli zawsze będziesz uruchamiał takie skrypty w tradycyjny sposób z poziomu wiersza poleceń, bez korzystania z wiersza *shebang*.

Jeżeli jednak używasz *Pythona w wersji 3.3 lub nowszej* lub jeśli osobno zainstalowałeś program uruchamiający (launcher) dla Pythona, to przekonasz się, że uniksowe wiersze `#!` również mogą działać w systemie Windows. Oprócz możliwości opisywanych wcześniej, nowy program uruchamiający `py` potrafi parsować wiersze *shebang* określające, która wersja Pythona ma zostać uruchomiona do wykonania skryptu. Ponadto `py` pozwala na podanie numeru wersji w pełnej lub częściowej formie i rozpoznaje popularne wzorce uniksowe stosowane w tym wierszu, włącznie z użyciem `/usr/bin/env`.

Mechanizm parsowania wiersza `#!` przez program uruchamiający jest używany po uruchomieniu skryptu z poziomu wiersza poleceń powłoki systemu za pomocą programu `py` oraz po kliknięciu ikony pliku zawierającego program w języku Python (wtedy program uruchamiający `py` jest wywoływany niejawnie przez skojarzenie rozszerzenia nazwy pliku). W przeciwieństwie do systemu Unix, w Windows nie musisz nadawać plikom uprawnień do wykonywania, ponieważ podobne rezultaty są osiągane dzięki skojarzeniom nazw plików.

Na przykład, pierwsze polecenie przedstawione poniżej jest uruchamiane przez Pythona 3.x, a drugie przez wersję 2.x (bez podawania konkretnego numeru program uruchamiający domyślnie przyjmuje wersję 2.x, chyba, że odpowiednio ustawisz zmienną środowiskową `PYTHON`):

```
c:\code> type robin3.py
#!/usr/bin/python3

print('Uciekaj', 'stąd!...')                      # funkcja dla wersji 3.x

c:\code> py robin3.py                                # Uruchomienie pliku z
użyciem wiersza #!

Uciekaj stąd!...

c:\code> type robin2.py
#!python2

print 'Uciekaj', 'stąd, szybko!...'                 # wyrażenie dla wersji 2.x

c:\code> py robin2.py                                # Uruchomienie pliku z użyciem
wiersza #!

Uciekaj stąd, szybko!...
```

Jak widać, numer wersji Pythona, przeznaczonej do uruchomienia danego pliku programu, można przekazywać nie tylko jako argument wywołania lauchera `py`, co pokazywaliśmy już wcześniej, ale również za pośrednictwem wiersza *shebang* — działa to w taki sam sposób:

```
c:\code> py -3.1 robin3.py                         # Uruchamianie z numerem wersji w
wierszu polecenia

Uciekaj stąd!...
```

W rezultacie program uruchamiający pozwala na określenie żądanej wersji Pythona zarówno na podstawie zawartości pliku (wiersz `#!`), jak i poprzez podanie odpowiedniego *argumentu wywołania* w wierszu polecenia. To była bardzo krótka opowieść o programie uruchamiającym. Jeżeli jeszcze tego nie zrobiłeś, a w systemie Windows używasz Pythona w wersji 3.3 lub

nowszej, lub będziesz takiej wersji używał w przyszłości, powinieneś zatrzymać się na dodatku B, gdzie znajdziesz wiele dodatkowych informacji na temat programu uruchamiającego.

Klikanie ikon plików

Jeżeli nie jesteś fanem wiersza poleceń, możesz go na ogół uniknąć, uruchamiając skrypty Pythona za pomocą klikania ikon plików, używając środowisk programistycznych wyposażonych w graficzne interfejsy użytkownika i korzystając z innych rozwiązań, które mogą się różnić w zależności od platformy. Rzućmy okiem na pierwszy z tych sposobów.

Podstawowe zagadnienia związane z klikaniem ikon plików

Klikanie ikon plików jest w takiej czy innej formie obsługiwane na większości platform wyposażonych w graficzne środowisko użytkownika. Oto krótki opis, jak może to funkcjonować na Twoim komputerze:

Klikanie ikon w systemie Windows

W systemie *Windows* otwieranie plików za pomocą klikania ich ikon jest łatwe — dzięki rejestrowi. Po zainstalowaniu Python wykorzystuje Windowsowy mechanizm *skojarzeń nazw plików* do automatycznego zarejestrowania się jako program otwierający pliki programów napisanych w tym języku. Dzięki temu możliwe jest uruchomienie programów napisanych w Pythonie przez proste kliknięcie (lub podwójne kliknięcie) ikon ich plików za pomocą myszy.

Kliknięty plik zostanie uruchomiony przez jeden z dwóch programów Pythona, w zależności od rozszerzenia nazwy pliku i używanej wersji Pythona. W wersji Python 3.2 i wcześniejszych, pliki *.py* są uruchamiane przez program *python.exe* w wersji konsolowej (wiersz poleceń), a pliki *.pyw* przez *pythonw.exe* (bez konsoli). Pliki kodu bajtowego po kliknięciu ikony są również uruchamiane przez te same programy. Tak jak to opisaliśmy w dodatku B, w Pythonie 3.3 i nowszych wersjach, nowe programy uruchamiające *py.exe* i *pyw.exe* w systemie Windows (oraz w systemach, w których zostały doinstalowane osobno) spełniają to samo zadanie, uruchamiając pliki z rozszerzeniami odpowiednio *.py* i *.pyw*.

Klikanie ikon w innych systemach

W systemach innych niż *Windows* najprawdopodobniej będziesz mógł zrobić to samo, jednak ikony, eksplorator plików, sposób nawigacji i inne elementy mogą się nieco różnić. Na przykład w systemie *macOS* możesz użyć programu *PythonLauncher*, który znajdziesz w folderze *Aplikacje/MacPython* (lub *Python N.M*).

W systemie *Linux* i niektórych innych systemach uniksowych być może konieczne będzie zarejestrowanie rozszerzenia *.py* za pomocą GUI eksploratora plików, nadanie skryptowi „wykonywalności” za pomocą omówionej wcześniej sztuczki ze znakami *#!* lub skojarzenie typu MIME pliku z aplikacją lub poleceniem za pomocą edycji odpowiednich plików konfiguracyjnych, instalacji programów czy z użyciem innych narzędzi. Więcej szczegółowych informacji znajdziesz w dokumentacji eksploratora plików danego systemu.

Innymi słowy, klikanie ikon działa na ogół tak, jakbyś tego oczekował na swojej platformie, ale pamiętaj, że warto zapoznać się z dokumentacją dla danej platformy; znajdziesz tam szczegółowe informacje na temat konfigurowania i używania Pythona.

Kliknięcie ikony w systemie Windows

Aby zilustrować tę kwestię, wykorzystamy utworzony wcześniej skrypt *script1.py*, którego kod jeszcze raz zamieszczamy poniżej w celu uniknięcia konieczności przewracania strony:

```
# Pierwszy skrypt w Pythonie

import sys                                # Załadowanie modułu biblioteki

print(sys.platform)
print(2 ** 100)                            # Podniesienie 2 do potęgi
x = 'Mielonka!'
print(x * 8)                             # Powtórzenie łańcucha znaków
```

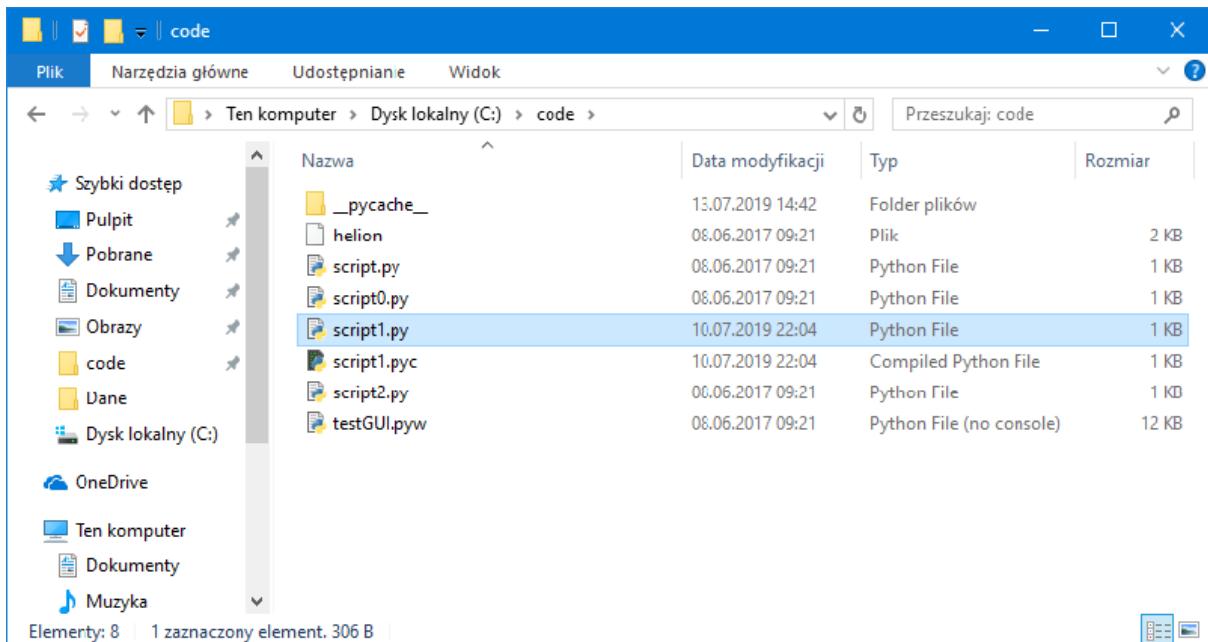
Jak już pokazywaliśmy wcześniej, plik *script1.py* można uruchomić za pomocą systemowego wiersza poleceń:

```
C:\code> python script1.py
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Kliknięcie ikony pliku pozwala nam jednak na uruchomienie go bez konieczności wpisywania czegokolwiek. Aby to zrobić, musisz znaleźć ikonę tego pliku na swoim komputerze. W systemie Windows 10 możesz kliknąć lewym przyciskiem myszy ikonę *Eksploratora plików*, znajdującą się na pasku zadań. We wcześniejszych wersjach systemu Windows możesz wybrać polecenie *Komputer* (lub *Mój komputer* w XP) w menu przycisku *Start*. Istnieją również inne metody otwierania eksploratora plików; kiedy już to zrobisz, przejdź na dysku *C* do katalogu roboczego.

Na ekranie powinieneś mieć okno eksploratora plików podobne do przedstawionego na rysunku 3.1 (w tym przykładzie używamy systemu Windows 10). Zwróć uwagę, jak wyglądają ikony plików Pythona:

- Pliki z kodem źródłowym mają jasne tło.
- Pliki z kodem bajtowym mają ciemne tło.



Rysunek 3.1. W systemie Windows pliki programów Pythona są wyświetlane w eksploratorze plików w postaci ikon i mogą automatycznie zostać uruchomione po ich dwukrotnym kliknięciu przyciskiem myszy (choć w ten sposób możemy nie zobaczyć na ekranie wyników działania czy komunikatów o błędach)

W poprzednim rozdziale utworzyłem plik kodu bajtowego widoczny na tym rysunku, importując źródła w Pythonie 3.1; wersje 3.2 i późniejsze zamiast tego przechowują pliki kodu bajtowego w podkatalogu `__pycache__`, który również został pokazany na rysunku (tym razem pliki kodu bajtowego zostały utworzone poprzez zimportowanie źródeł w Pythonie 3.3). W zdecydowanej większości przypadków będziesz klikać (lub uruchamiać w inny sposób) pliki kodu źródłowego, aby mieć pewność, że uruchomiona zostanie najnowsza wersja programu — jeżeli zamiast tego uruchamiasz pliki kodu bajtowego, pamiętaj, że Python nie sprawdzi pliku kodu źródłowego pod kątem zmian. Aby zatem uruchomić tutaj nasz skrypt, po prostu dwukrotnie kliknij ikonę pliku `script1.py`.

Sztuczka z funkcją input

Niestety, w systemie Windows rezultat kliknięcia ikony pliku może nie być szczególnie satysfakcjonujący. Tak naprawdę powyższy przykładowy skrypt może spowodować po kliknięciu (uruchomieniu) denerwujące „mignięcie” — z pewnością nie jest to ten rodzaj informacji zwrotnych, na jakie liczą początkujący programiści Pythona! Nie jest to błąd, ma jednak swoją przyczynę w sposobie, w jaki wersja Pythona przeznaczona dla systemu Windows radzi sobie z wyświetlaniem wyników działania.

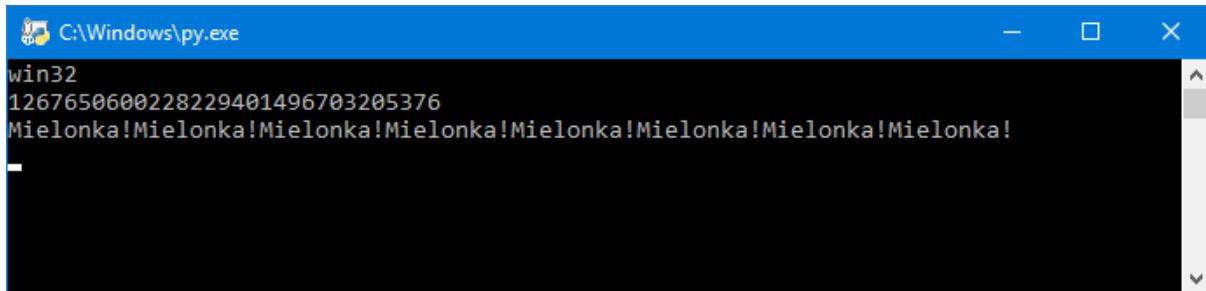
Domyślnie Python otwiera wyskakujące czarne okno konsoli DOS (*Wiersz poleceń*), które służy jako wejście i wyjście dla klikniętego pliku. Jeżeli skrypt po prostu wyświetla coś i kończy działanie, wtedy pokazuje się okno konsoli, w którym wyświetlany jest tekst, a po zakończeniu działania programu okno konsoli zamknięte się. O ile nie jesteś bardzo szybki albo Twój komputer nie jest bardzo wolny, nie dasz rady w ogóle zobaczyć wyników działania programu. Choć jest to normalne zachowanie programu, to raczej nie o to Ci chyba chodziło.

Na szczęście istnieje prosty sposób na ominięcie tego problemu. Jeżeli chcesz zachować wyniki działania pliku uruchomionego za pomocą kliknięcia ikony, wystarczy w ostatnim wierszu

skryptu umieścić wywołanie wbudowanej funkcji `input` (w wersji 2.x była to funkcja `raw_input`; zobacz wskazówkę poniżej). Na przykład:

```
# Pierwszy skrypt w Pythonie
import sys                                # Załadowanie modułu biblioteki
print(sys.platform)
print(2 ** 100)                            # Podniesienie 2 do potęgi
x = 'Mielonka!'
print(x * 8)                             # Powtórzenie łańcucha znaków
input()                                    # <== DODANA FUNKCJA
```

Generalnie funkcja `input` wczytuje kolejny wiersz danych ze standardowego wejścia i czeka, jeżeli żaden wiersz nie jest dostępny. Rezultat będzie więc taki, że działanie skryptu zostanie zatrzymane, tym samym zachowując na ekranie okno z wynikami działania, widoczne na rysunku 3.2. Okno zostanie zamknięte dopiero po naciśnięciu klawisza *Enter*.



Rysunek 3.2. Kiedy w systemie Windows klikamy ikonę programu, wyniki działania programu będziemy mogli zobaczyć po dodaniu wywołania funkcji `input()` na końcu skryptu. Z tej sztuczki powinieneś jednak korzystać tylko w tej jednej sytuacji!

Skoro już pokazałem tę sztuczkę, warto podkreślić, że zazwyczaj jest ona wymagana jedynie w systemie Windows — tylko wtedy, gdy nasz skrypt wyświetla jakiś tekst i kończy działanie, oraz gdy uruchamiamy plik, klikając jego ikonę. Dodatkowe wywołanie funkcji `input` powinieneś się dodawać w ostatnim wierszu plików najwyższego poziomu tylko i wyłącznie wtedy, gdy spełnione są wszystkie trzy powyższe warunki. Nie ma sensu dodawać go w innych scenariuszach, takich jak użycie skryptów uruchamianych bezpośrednio z poziomu konsoli czy skryptów działających w środowisku graficznym IDLE (chyba że lubisz sobie od czasu do czasu naciskać bez potrzeby klawisz *Enter*)^[2]. Może się to wydawać oczywiste, jednak jest to kolejny, często popełniany błąd.

Zanim przejdziemy dalej, warto zwrócić uwagę na to, że zastosowane tutaj wywołanie funkcji `input` jest dla wejścia odpowiednikiem tego, co w przypadku wyjścia robi się za pomocą funkcji `print` (lub polecenia `print` w wersjach 2.x). To najprostszy sposób pobierania danych wejściowych od użytkownika i jest on o wiele bardziej uniwersalny, niż sugeruje to nasz prosty przykład. Przykładowo funkcja `input`:

- opcjonalnie przyjmuje łańcuch znaków, który zostanie wyświetlony jako zachęta (na przykład `input('Naciśnij klawisz Enter, aby zakończyć działanie programu')`),
- zwraca do skryptu pobrany wiersz tekstu jako ciąg znaków (na przykład `nextinput = input()`),

- obsługuje przekierowania strumienia wejścia na poziomie powłoki systemowej (na przykład `python spam.py < input.txt`) — podobnie jak instrukcja `print` robi to dla wyjścia.

Funkcję `input` będziemy wykorzystywać w nieco bardziej zaawansowany sposób w dalszej części tekstu; na przykład w rozdziale 10. będziemy jej używać w interaktywnej pętli, ale teraz przyjmij po prostu, że używamy jej do zatrzymania na ekranie wyników działania skryptów uruchamianych za pomocą podwójnego kliknięcia ikony pliku.

	<p><i>Uwagi na temat wersji:</i> Jeżeli pracujesz w Pythonie 2.x, powinieneś w powyższym kodzie w miejsce funkcji <code>input()</code> użyć <code>raw_input()</code>. Funkcja <code>raw_input()</code> zmieniła w Pythonie 3.x nazwę na <code>input()</code>. Z technicznego punktu widzenia w Pythonie 2.x również istnieje funkcja <code>input</code>, która jednak przeznaczona jest do analizowania łańcuchów znaków tak, jakby były one kodem programu osadzonym skrypcie, przez co nie zadziała ona w tym kontekście (pusty łańcuch znaków spowoduje błąd). Funkcja <code>input()</code> z Pythona 3.x (oraz <code>raw_input()</code> z wersji 2.x) zwraca po prostu wprowadzony tekst jako ciąg znaków, bez analizowania go. Aby uzyskać w Pythonie 3.x efekt funkcji <code>input</code> z wersji 2.x, powinieneś użyć wywołania <code>eval(input())</code>.</p> <p>Pamiętaj jednak, że skoro przedstawiona funkcja wykonuje wprowadzony tekst tak, jakby był <i>kodem programu</i>, może to mieć poważne implikacje dla bezpieczeństwa, które tutaj jednak w dużej mierze zignorujemy. Ograniczymy się tylko do stwierdzenia, że powinieneś używać zaufanego źródła wprowadzanego tekstu; jeżeli tego nie zrobisz, powinieneś się trzymać zwykłej funkcji <code>input</code> w wersji 3.x i <code>raw_input</code> w 2.x.</p>
---	---

Inne ograniczenia programów uruchamianych kliknięciem ikony

Nawet z zastosowaniem sztuczki z funkcją `input`, klikanie ikon plików ma swoje ograniczenia. Możemy na przykład nie zobaczyć komunikatów o błędach Pythona. Jeżeli skrypt wygeneruje błąd, komunikat o błędzie wyświetlany jest w wyskakującym oknie konsoli — które natychmiast potem znika! Co gorsza, dodanie wywołania funkcji `input` do skryptu tym razem nie pomoże, ponieważ zakończy on działanie, zanim dojdzie do tego wywołania. Innymi słowy, nie będziesz w stanie ustalić, co poszło nie tak.

Kiedy w dalszej części tej książki będziemy omawiać wyjątki, przekonasz się, że można tworzyć kod, który będzie przechwytywał, przetwarzał i obsługiwał błędy tak, aby nie przerywały działania programów. Zwróć uwagę na omówienie instrukcji `try` w dalszej części tej książki, gdzie pokażemy co powinieneś zrobić, aby okno konsoli nie zamknęło się po wystąpieniu błędu. Podczas omawiania funkcji `print` dowiesz się również, jak przekierować wyświetlany tekst do plików w celu późniejszego sprawdzenia. Pomięcie obsługi błędów w kodzie może spowodować, że komunikaty o błędach, wyświetlane przez programy uruchamiane za pośrednictwem kliknięcia ikony pliku, zostaną utracone.

Ze względu na te ograniczenia najlepiej jest chyba potraktować klikanie ikon plików jako sposób uruchamiania programów stosowany dopiero po ich gruntownym przetestowaniu i usunięciu błędów lub po przekierowaniu wyników działania do pliku. Jeżeli jesteś początkującym użytkownikiem Pythona, powinieneś raczej korzystać z innych sposobów uruchamiania programów — na przykład systemowego wiersza poleceń czy środowiska IDLE (więcej szczegółowych informacji na ten temat znajdziesz w dalszej części rozdziału, w sekcji zatytułowanej „Interfejs użytkownika środowiska IDLE”) — tak by nie przegapić wygenerowanych komunikatów o błędach i zobaczyć wyniki działania programu bez uciekania się do tanich sztuczek.

Importowanie i przeładowywanie modułów

Dotychczas mówiliśmy o importowaniu modułów bez dokładniejszego wyjaśniania, co ten termin oznacza. Moduły oraz architekturę programów omówimy bardziej szczegółowo w V części książki, jednak ze względu na fakt, że importowanie jest również jednym ze sposobów uruchamiania programów, w tym podrozdziale omówimy podstawowe zagadnienia związane z modułami, co pozwoli Ci z nich korzystać już teraz.

Podstawy importowania i przeładowywania modułów

Mówiąc w uproszczeniu, każdy plik z kodem źródłowym Pythona, którego nazwa kończy się rozszerzeniem `.py`, jest modułem. Aby plik kodu źródłowego stał się modułem, nie jest wymagany żaden specjalny kod ani składnia; po prostu każdy taki plik jest modułem. Inne pliki mogą uzyskać dostęp do elementów modułu, *importując* ten moduł — operacja importu pozwala na załadowanie innego pliku i daje dostęp do jego zawartości. Zawartość modułu jest udostępniana poprzez jego atrybuty (o czym powiemy już za chwilę).

Taki oparty na modułach model usług okazuje się być podstawową koncepcją architektury programów w języku Python. Większe programy zazwyczaj składają się z większej liczby plików modułów, które importują narzędzia z innych plików modułów. Jeden z modułów staje się plikiem głównym lub — inaczej mówiąc — „skryptem”, czyli plikiem, który używany jest do uruchomienia całego programu. Wszystkie inne pliki podlegające to po prostu moduły, które mogą importować dalsze moduły.

Zagadnieniem architektury programów zajmiemy się bardziej szczegółowo w dalszej części książki. W tym rozdziale najbardziej interesuje nas fakt, że operacje importowania *uruchamiają* kod w pliku ładowanym jako ostatni. Z tego powodu możemy powiedzieć, że importowanie jest kolejnym sposobem uruchamiania pliku.

Na przykład, jeżeli rozpocznesz sesję interaktywną (z systemowego wiersza poleceń, z menu *Start*, środowiska IDLE czy w dowolny, inny sposób), możesz uruchomić wcześniej utworzony plik `script1.py` za pomocą prostej instrukcji `import` (warto jednak pamiętać o wcześniejszym usunięciu wiersza z wywołaniem funkcji `input`, dodanego w poprzednim podrozdziale, gdyż inaczej będziesz musiał niepotrzebnie nacisnąć klawisz `Enter`):

```
C:\code> c:\python33\python
>>> import script1
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Takie coś działa, ale domyślnie tylko raz na sesję (a tak naprawdę raz na *proces* — uruchomienie programu). Po pierwszym zimportowaniu kolejne nie robią już nic, nawet jeśli w innym oknie plik źródłowy modułu zmienimy i zapiszemy ponownie:

```
...Otwórz plik script1.py do edycji i zmień kod tak, aby wyświetlał wynik
działania 2 ** 16
>>> import script1
>>> import script1
```

Takie zachowanie jest celowe — importowanie jest zbyt kosztowną operacją, by powtarzać ją częściej niż raz na plik czy na uruchomienie programu. Jak się przekonasz w rozdziale 22., operacja `import` musi odnaleźć pliki, skompilować je do kodu bajtowego i dopiero wtedy może je wykonać.

Jeżeli naprawdę chcesz zmusić Pythona do ponownego wykonania pliku w tej samej sesji (bez jej zatrzymania i ponownego uruchomienia), musisz zamiast tego wywołać wbudowaną funkcję `reload`, dostępną w module standardowej biblioteki `imp` (funkcja ta dostępna jest także w Pythonie 2.x jako zwykła funkcja wbudowana, jednak w 3.x tak już nie jest):

```
>>> from imp import reload          # W wersji 3.x trzeba załadować z modułu
>>> reload(script1)
win32
65536
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
<module 'script1' from '.\\script1.py'>
>>>
```

Wykorzystana powyżej instrukcja `from` po prostu kopiuje nazwę z modułu (więcej na ten temat wkrótce). Sama funkcja `reload` ładuje i wykonuje aktualną wersję kodu pliku, uwzględniając tym samym wszelkie zmiany, które mogły być wprowadzone i zapisane w osobnym oknie edytora.

Pozwala to na edytowanie i testowanie nowego kodu w locie w czasie bieżącej sesji interaktywnej Pythona. Na przykład w naszej sesji drugie polecenie `print` w pliku `script1.py` zostało między pierwszym importem a wywołaniem funkcji `reload` zmienione na `print 2 ** 16` — stąd inny wynik działania programu.

Funkcja `reload` oczekuje podania nazwy już załadowanego modułu, więc zanim przeładowanie będzie możliwe, moduł musi zostać poprawnie zimportowany co najmniej raz (jeżeli podczas importowania wystąpił błąd, przeładowanie nie będzie możliwe i trzeba będzie taki moduł zimportować ponownie). Warto zauważyć, że funkcja `reload` oczekuje również ujęcia nazwy modułu w nawiasy, natomiast `import` nie ma takich wymagań. Funkcja `reload` jest funkcją wywoływaną, natomiast `import` jest instrukcją.

Wywołując funkcję `reload` przekazujemy do niej nazwę modułu, a w wynikach działania pojawia się dodatkowy wiersz — ostatni wiersz wyników zawiera wartość zwracaną przez funkcję `reload`, czyli obiekt modułu Pythona. Więcej informacji na temat wykorzystywania funkcji znajdziesz w rozdziale 16.; teraz wystarczy, że kiedy usłyszysz „funkcja”, będziesz wiedział, że jej argumenty trzeba umieścić w nawiasach.



Uwagi na temat wersji: W Pythonie 3.x przesunięto wbudowaną funkcję `reload` do modułu standardowej biblioteki `imp`. Działa ona tak jak do tej pory, z tym że teraz, aby z niej korzystać, musisz ją najpierw zimportować. W wersji 3.x powinieneś wykonać polecenie `import imp` i wywołać funkcję `imp.reload(M)` lub wykonać polecenie `from imp import reload` i wywołać funkcję `reload(M)`, jak pokazano w przykładzie powyżej. O poleceniach `import` oraz `from` opowiem jeszcze w kolejnym podrozdziale, a bardziej formalnie omówimy je w dalszej części książki.

Jeżeli pracujesz w Pythonie 2.x, przeładowanie modułu dostępne jest w postaci wbudowanej funkcji `reload`, dlatego jej importowanie nie jest konieczne. W wersjach 2.6 i 2.7 funkcja `reload` dostępna jest w *obu* formach — funkcji wbudowanej oraz funkcji modułu — co miało ułatwić przejście do wersji 3.x. Innymi słowy, przeładowywanie modułów jest nadal dostępne w wersji 3.x,

jednak do wywołania funkcji `reload` niezbędny jest dodatkowy wiersz kodu pobierający tę funkcję z modułu biblioteki.

Przeniesienie funkcji `reload` do biblioteki w Pythonie 3.x spowodowane było po części pewnymi znanimi problemami związanymi z użyciem funkcji `reload` i instrukcji `from`, z którymi spotkamy się w kolejnym podrozdziale. Mówiąc w skrócie, nazwy załadowane za pomocą instrukcji `from` nie są bezpośrednio aktualizowane przez przeładowanie, ale nazwy dostępne za pomocą polecenia `import` już tak. Jeżeli nazwy nie zmieniają się po ponownym załadowaniu modułu, spróbuj zamiast tego użyć polecenia `import` i odwołań w postaci `moduł.atrybut`.

Więcej o modułach – atrybuty

Importowanie i przeładowywianie modułów stanowi naturalną opcję uruchamiania, ponieważ ostatnim etapem działania operacji `import` jest wykonanie pliku. W szerszym kontekście moduły pełnią jednak rolę bibliotek narzędzi, co zostanie szczegółowo omówione w piątej części książki. Podstawowa idea jest prosta — moduł jest po prostu pakietem nazw zmiennych, znanych jako *przestrzeń nazw* (ang. *namespace*). Nazwy w tym pakiecie nazywane są *atrybutami* — atrybut jest po prostu nazwą zmiennej, która jest przywiązana do określonego obiektu (takiego jak moduł).

W typowym scenariuszu użycia plik importujący zyskuje dostęp do wszystkich nazw (zmiennych) przypisanych na najwyższym poziomie pliku modułu. Te nazwy są zazwyczaj przypisywane narzędziom eksportowanym przez moduł — funkcjom, klasom, zmiennym i tym podobnym — które mają być wykorzystane przez inne pliki czy programy. Nazwy z pliku modułu można pobrać z zewnątrz za pomocą dwóch instrukcji Pythona — `import` i `from` — a także dzięki wywołaniu funkcji `reload`.

Aby to zilustrować, wykorzystamy edytor tekstu do utworzenia w katalogu roboczym krótkiego, jednowierszowego pliku modułu o nazwie `myfile.py`, zawierającego następujący kod:

```
title = "Sens życia"
```

Być może jest to jeden z najprostszych modułów Pythona na świecie (zawiera tylko jedną instrukcję przypisania), jednak wystarczy nam on do zilustrowania podstaw. Kiedy zaimportujemy ten plik, jego kod jest wykonywany, by wygenerować atrybuty modułu. Oznacza to, że instrukcja przypisania tworzy zmienną i atrybut modułu o nazwie `title`.

Dostęp do atrybutu `title` tego modułu można z innych komponentów uzyskać na dwa sposoby. Po pierwsze, można załadować moduł jako całość za pomocą instrukcji `import`, a następnie poprzedzić nazwę atrybutu kropką i nazwą modułu (nazywane jest to czasem *kwalifikacją*):

```
% python # Uruchomienie Pythona
>>> import myfile # Wykonanie pliku; moduł ładowany jest w
całości
>>> myfile.title # Wykorzystanie nazw atrybutów: użycie znaku
.
.
.
Sens życia
```

Generalnie składnia z kropką — w postaci `obiekt.atrybut` — pozwala na pobranie dowolnego atrybutu dołączonego do dowolnego obiektu i w kodzie Pythona taka operacja wykonywana jest bardzo często. W powyższym przykładzie wykorzystaliśmy ten zapis do uzyskania dostępu do zmiennej `title` znajdującej się w module `myfile` — inaczej mówiąc, `myfile.title`.

Alternatywnie można pobrać (a tak naprawdę skopiować) nazwy z modułu za pomocą instrukcji `from`:

```
% python                                     # Uruchomienie Pythona
>>> from myfile import title                 # Wykonanie pliku; skopiowanie jego nazw
>>> title                                    # Bezpośrednie wykorzystanie nazw: nie ma
konieczności
                                                # użycia znaku '.' i nazwy modułu
```

Sens życia

Jak zobaczymy w szczegółach nieco później, instrukcja `from` jest podobna do `import`, jednak dodatkowo przypisuje nazwy w importowanych komponentach. Z technicznego punktu widzenia instrukcja ta kopiuje *atrybuty* modułu w taki sposób, że stają się one *zmiennymi* w kodzie importującym. Dzięki temu do zaimportowanego łańcucha znaków można się tym razem odnosić za pomocą samego `title` (zmiennej), a nie `myfile.title` (odniesienia do atrybutu modułu)^[3].

Bez względu na to, czy do wywołania operacji importowania użyjemy polecenia `import`, czy `from`, instrukcje z modułu `myfile.py` są wykonywane, a komponent importujący (tutaj sesja interaktywna) uzyskuje dostęp do nazw przypisanych na najwyższym poziomie pliku. W tym prostym przykładzie znajduje się tylko jedna taka nazwa — zmienna `title` przypisana do łańcucha znaków — jednak sama koncepcja stanie się o wiele bardziej użyteczna, kiedy zaczniemy w modułach definiować obiekty, takie jak funkcje czy klasy. Takie obiekty staną się komponentami oprogramowania, które można wykorzystać ponownie i do których dostęp z innych modułów klienta odbywa się za pomocą wywołania ich nazwy.

W praktyce pliki modułów definiują zazwyczaj więcej niż jedną nazwę, która jest następnie wykorzystywana w tym pliku, a także poza nim. Poniżej znajduje się przykład modułu definiującego trzy zmienne:

```
a = 'skecz'                                # Zdefiniowanie trzech atrybutów
b = 'z martwą'                               # Wyeksportowanie ich do innego pliku
c = 'papugą'
print(a, b, c)                               # Wykorzystanie ich także w tym pliku (w wersji 2.x:
print a, b, c)
```

Ten plik (`threenames.py`) przypisuje trzy zmienne i tym samym generuje trzy atrybuty dla plików zewnętrznych. Trzy swoje zmienne wykorzystuje również funkcja `print` (dla wersji 3.x), co widać po wykonaniu tego pliku jako pliku najwyższego poziomu (w Pythonie 2.x składnia polecenia `print` różni się nieznacznie — wystarczy w wywołaniu funkcji pominąć nawiasy, aby otrzymać dokładnie taki sam wynik; więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 11.):

```
% python threenames.py
skecz z martwą papugą
```

Kod ten zostanie wykonany, kiedy tylko zostanie pierwszy raz zimportowany w innym miejscu (za pomocą instrukcji `import` lub `from`). Pliki importujące ten kod za pomocą polecenia `import` otrzymują moduł z atrybutami, natomiast te wykorzystujące instrukcję `from` dostają kopie nazw z oryginalnego pliku:

```
% python
>>> import threenames                         # Pobranie całego modułu; zostaje
tutaj uruchomiony.
skecz z martwą papugą
```

```

>>>
>>> threenames.b, threenames.c          # Dostęp do atrybutów
('z martwą', 'papugą')
>>>
>>> from threenames import a, b, c      # Skopiowanie wybranych nazw
>>> b, c
('z martwą', 'papugą')

```

Wyniki wyświetlane są w nawiasach, ponieważ tak naprawdę są *krotkami*, czyli rodzajem obiektów składających się z szeregu elementów ujętych w nawiasy okrągłe i oddzielonych od siebie przecinkami (bardziej szczegółowo omówimy je w dalszej części książki); kwestię tę możemy na razie zignorować.

Kiedy zaczniemy tworzyć moduły zawierające po kilka nazw, przyda nam się wbudowana funkcja `dir`. Można jej użyć do pobrania listy nazw dostępnych w module. Poniższy kod zwraca pythonową listę ciągów znaków, ujętą w nawiasy kwadratowe (listy zaczniemy omawiać w kolejnym rozdziale):

```

>>> dir(threenames)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'a', 'b',
 'c']

```

Zawartość tej listy została tutaj edytowana, ponieważ może się różnić w zależności od wersji Pythona. Kiedy wywołuje się funkcję `dir` i w nawiasach przekazuje się jej nazwę importowanego modułu, zwraca ona wszystkie atrybuty znajdujące się w tym module. Niektóre ze zwracanych nazw dostajemy „gratis” — nazwy z początkowymi i końcowymi podwójnymi znakami podkreślenia (`_X_`) są wbudowane, predefiniowane w Pythonie i mają specjalne znaczenie dla interpretera, ale w tym miejscu książki nie są jeszcze dla nas ważne. Zmienne zdefiniowane w naszym kodzie przez przypisanie (`a`, `b` oraz `c`) pokazują się na końcu listy będącej wynikiem wywołania funkcji `dir`.

Moduły i przestrzenie nazw

Importowanie modułów to sposób wykonywania kodu z pliku, jednak — jak pokażemy w dalszej części książki — moduły to również największe struktury w programach napisanych w Pythonie i jedne z pierwszych, kluczowych pojęć w tym języku.

Generalnie programy w Pythonie składają się z pewnej liczby plików modułów połączonych ze sobą za pomocą instrukcji `import`. Każdy plik modułu jest samodzielnym pakietem zmiennych, czyli *przestrzenią nazw*. Co równie ważne, każdy moduł jest *samodzielną* przestrzenią nazw: jeden plik modułu nie może widzieć nazw zdefiniowanych w innym pliku, dopóki w jawnym sposób nie zimportuje tego pliku, dlatego moduły służą do zapobiegania *konfliktom nazw* w kodzie. Ponieważ każdy plik jest samodzielną przestrzenią nazw, nazwy z jednego pliku nie mogą pozostać w konflikcie z nazwami z innych plików, nawet jeśli zapisane są w ten sam sposób.

Tak naprawdę moduły to jeden z kilku sposobów, w jaki Python dzieli zmienne na pakiety w celu uniknięcia konfliktu nazw. Moduły oraz inne konstrukcje przestrzeni nazw (wraz z klasami i zakresem funkcji) omówione zostaną w dalszej części książki. Na razie przydadzą się nam jako sposób na wielokrotne wykonywanie kodu bez konieczności wpisywania go ponownie oraz na zapobieganie przypadkowemu zastępowaniu nazw plików.

	<i>Instrukcja import kontra from:</i> Powinienem wspomnieć, że instrukcja <code>from</code> w pewnym sensie niweluje podział przestrzeni nazw wprowadzany przez moduły — dzieje się tak, ponieważ <code>from</code> kopiuje zmienne z jednego pliku do drugiego,
--	--



może powodować nadpisywanie zmiennych o tych samych nazwach w pliku importującym (i to bez żadnego ostrzeżenia). Zasadniczo, powoduje to właściwie złączenie ze sobą przestrzeni nazw, przynajmniej w zakresie skopiowanych zmiennych.

Z tego powodu niektórzy użytkownicy zalecają wykorzystywanie instrukcji `import` w miejsce `from`. Ja sam nie pójdę aż tak daleko — po pierwsze, `from` wymaga mniej pisania (co jest sporą zaletą w sesjach interaktywnych), a po drugie, ten potencjalny problem w praktyce pojawi się dość rzadko. Poza tym jest to coś, co możemy sami kontrolować, wymieniając zmienne w instrukcji `from`. Dopóki rozumiemy, że zmienne zostaną przypisane do wartości w module docelowym, nie jest to bardziej niebezpieczne od tworzenia instrukcji przypisania — kolejnej opcji, z której pewnie będziesz chciał skorzystać.

Uwagi praktyczne — instrukcje `import` i `reload`

Z jakiegoś powodu, kiedy ludzie dowiadują się o możliwości wykonywania plików za pomocą instrukcji `import` i `reload`, wielu z nich skupia się na tej możliwości i zapomina o innych opcjach uruchamiania, które zawsze wykonują aktualną wersję kodu (czyli o klikaniu ikon, opcjach z menu środowiska IDLE, a także systemowym wierszu poleceń). Może to jednak szybko prowadzić do zamieszania — trzeba pamiętać o importowaniu, zanim będzie można przeładować moduł, o używaniu nawiasów tylko przy instrukcji `reload`, a przede wszystkim o samej konieczności używania `reload` w celu uzyskania dostępu do aktualnej wersji kodu. Co więcej, przeładowania nie są przechodnie — przeładowanie jednego modułu powoduje jedynie przeładowanie tego modułu, a nie wszystkich pozostałych, które może on importować, dlatego czasami konieczne jest wykonanie tej operacji dla większej liczby plików.

Ze względu na te utrudnienia (i inne, które omówimy później), włącznie z problemem z instrukcjami `reload` oraz `from` przedstawionym we wskazówce w poprzedniej sekcji, lepiej jest na razie oprzeć się pokusie uruchamiania poprzez importowanie i przeładowywianie modułów. Uruchamianie plików za pomocą opisanego w kolejnym podrozdziale menu środowiska IDLE (*Run/Run Module*) jest przykładem prostszego i mniej podatnego na błędy sposobu ich wykonywania, a ponadto zawsze wykonuje bieżącą wersję kodu. Podobne zalety oferuje systemowy wiersz poleceń. Przy korzystaniu z tych rozwiązań nie jest konieczne używanie instrukcji `reload`.

Dodatkowo, jeżeli w tym miejscu książki użyjemy modułów w nietypowy sposób, możemy popaść w tarapaty. Na przykład, jeżeli będziesz chciał zaimportować plik modułu przechowywany w katalogu innym od bieżącego, będziesz musiał przeskoczyć do rozdziału 22., w którym omówiona jest ścieżka wyszukiwania modułów. Na razie, jeżeli już musisz coś importować, w celu uniknięcia komplikacji lepiej będzie zachować wszystkie pliki w katalogu, w którym pracujesz^[4].

Z drugiej strony, w praktyce okazuje się jednak, że importowanie i przeładowywianie modułów całkiem nieźle sprawdza się przy testowaniu programów i możesz również preferować takie podejście. Pamiętaj jednak, że jeżeli ciągle potykasz się o te same kamienie, to po prostu trzeba zacząć je omijać.

Wykorzystywanie funkcji `exec` do wykonywania plików modułów

Tak naprawdę istnieje więcej sposobów uruchamiania kodu przechowywanego w plikach modułów, niż tylko te, które do tej pory omówiliśmy. Przykładowo wywołanie wbudowanej

funkcji `exec(open('module.py').read())` jest innym sposobem uruchamiania plików z sesji interaktywnej bez konieczności importowania i późniejszego przeładowywania ich. Każde wywołanie funkcji `exec` wykonuje aktualną wersję pliku, bez konieczności późniejszego przeładowywania go (plik `script1.py` jest obecnie w takiej formie, w jakiej go pozostawiliśmy po przeładowaniu w poprzednim podrozdziale):

```
% python
>>> exec(open('script1.py').read())
win32
65536
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
...zmodyfikuj plik script1.py w edytorze tekstu tak, aby wyświetlał wynik
operacji 2 ** 32...
>>> exec(open('script1.py').read())
win32
4294967296
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Funkcja `exec` daje efekt podobny do importowania, jednak z technicznego punktu widzenia nie importuje modułu — domyślnie za każdym wywołaniem tej funkcji wykonuje ona plik od nowa, tak jakbyśmy wkleili jego kod w miejsce, gdzie wywołuje się `exec`. Z tego powodu funkcja ta nie wymaga przeładowywania modułu po modyfikacji pliku — pomija ona normalną logikę importowania modułów.

Wadą tego rozwiązania jest to, że funkcja `exec`, przypominając w swym działaniu wklejanie kodu do miejsca, w którym jest wywołana — podobnie do wspomnianej wcześniej instrukcji `from` — może potencjalnie po cichu nadpisać zmienne, których aktualnie używamy. Przykładowo nasz skrypt `script1.py` przypisuje wartość do zmiennej o nazwie `x`. Jeżeli nazwa ta wykorzystywana jest także w miejscu, w którym wywołujemy funkcję `exec`, jej wartość zostanie zastąpiona:

```
>>> x = 999
>>> exec(open('script1.py').read())      # Kod domyślnie wykonywany jest w tej
przestrzeni nazw
...te same wyniki działania...
>>> x                                # Przypisania w programie mogą tutaj
nadpisać istniejące zmienne
'Mielonka!'
```

Podstawowe polecenie `import` wykonuje z kolei plik tylko raz na proces i nadaje mu osobną przestrzeń nazw modułu, tak by przypisania nie zmieniły zmiennych w bieżącym zakresie. Ceną za podział przestrzeni nazw modułów jest konieczność przeładowywania ich po wprowadzeniu zmian.



Uwaga na temat wersji: W Pythonie 2.x poza wywołaniem `exec(open('module.py'))` możemy również użyć wbudowanej funkcji `execfile('module.py')`; obie automatycznie wczytują zawartość pliku. Obydwie postacie są równoważne z wywołaniem funkcji `exec(open('module.py').read())`, które jest nieco bardziej skomplikowane, natomiast działa zarówno w wersji 2.x, jak i 3.x.

Niestety, żadna z prostszych form wywołania z Pythona 2.x nie jest dostępna w wersji 3.x, co oznacza, że obecnie, aby w pełni zrozumieć tę technikę, należy przyswoić sobie zarówno zagadnienia związane z plikami, jak i metodami ich wczytywania (wygląda to w istocie, jakby w wersji 3.x estetyka wygrała z praktycznością). Właściwie postać funkcji exec z Pythona 3.x wymaga tak dużo pisania, że najlepszym wyjściem wydaje się unikanie jej — zazwyczaj najlepiej jest uruchamiać pliki, wpisując odpowiednie polecenia w systemowym wierszu poleceń lub korzystając z opcji menu środowiska IDLE, opisanych w kolejnym podrozdziale.

Więcej informacji na temat interfejsów plików wykorzystywanych przez funkcję exec z Pythona 3.x znajdziesz w rozdziale 9. Więcej szczegółowych informacji na temat funkcji exec i powiązanych z nią funkcji eval i compile zamieściłem w rozdziałach 10. i 25.

Interfejs użytkownika środowiska IDLE

Dotychczas widzieliśmy, jak można uruchomić programy napisane w Pythonie za pomocą sesji interaktywnej, systemowego wiersza poleceń, klikania ikon, importowania modułów oraz wywołania funkcji exec. Osobom szukającym czegoś bardziej wizualnego przyda się środowisko IDLE, wyposażone w graficzny interfejs użytkownika (GUI), które jest standardową i darmową częścią systemu Pythona. Zazwyczaj określa się go mianem *zintegrowanego środowiska programistycznego* (ang. *integrated development environment, IDE*), ponieważ łączy on różne zadania programistyczne w jeden widok^[5].

W skrócie mówiąc, IDLE to środowisko programistyczne wyposażone w graficzny interfejs użytkownika, które pozwala na edycję, wykonywanie, przeglądanie i debugowanie programów napisanych w Pythonie. Co więcej, IDLE to przenośny kod napisany w Pythonie, dzięki czemu działa na większości platform, w tym Microsoft Windows, X Windows (dla platform Linux, Unix i podobnych do Uniksa), a także macOS (zarówno Classic, jak i OS X). Dla wielu osób IDLE jest łatwą w użyciu alternatywą dla wiersza poleceń, stanowiącą rozwiązanie daleko mniej podatne na problemy niż klikanie ikon i jednocześnie będącą świetnym sposobem dla początkujących użytkowników na samodzielne rozpoczęcie edycji i uruchamiania kodu. Korzystanie ze środowiska IDLE powoduje jednak, że ceną za większą wygodę pracy jest nieco mniejsza kontrola nad pewnymi aspektami uruchamiania czy działania tworzonych programów, ale zazwyczaj staje się to bardziej istotne dopiero później, gdy nabierzesz już dużego doświadczenia w programowaniu w Pythonie.

Szczegóły uruchamiania środowiska IDLE

Większość Czytelników powinna mieć możliwość natychmiastowego użycia środowiska IDLE, ponieważ jest to dzisiaj standardowy komponent w systemie macOS i większości instalacji Linuksa, a w systemie Windows jest instalowany automatycznie w standardowym pakiecie Pythona. Ponieważ specyfika tych platform jest różna, przed uruchomieniem tego środowiska warto omówić kilka wskazówek.

Technicznie rzecz biorąc, IDLE jest programem Pythona, który używa standardowej biblioteki tkinter GUI (w Pythonie 2.x nosi ona nazwę Tkinter) do tworzenia graficznego interfejsu użytkownika. To sprawia, że środowisko IDLE jest przenośne, czyli działa tak samo na wszystkich głównych platformach, ale oznacza również, że do używania tego środowiska w Twoim Pythonie musi być zaimplementowana obsługa biblioteki tkinter. Jest to co prawda standardem w systemach Windows, macOS i Linux, ale w niektórych systemach istnieją pewne

ograniczenia, a uruchamianie może się różnić w zależności od systemu. Oto kilka porad dotyczących poszczególnych platform:

- W systemie *Windows 7* i wcześniejszych uruchomienie środowiska IDLE jest łatwe, ponieważ po zainstalowaniu Pythona jest ono zawsze dostępne i posiada odpowiednie polecenie w menu przycisku *Start* (zobacz rysunek 2.1 w poprzednim rozdziale). Możesz je także uruchomić, chwytając prawym przyciskiem myszy ikonę pliku programu i przeciągając na ikonę pliku *idle.pyw*, znajdującego się w podkatalogu *Lib/idlelib* Pythona. W tym przypadku środowisko IDLE to po prostu skrypt Pythona, zlokalizowany w katalogu *C:\Python33\Lib\idlelib*, *C:\Python27\Lib\idlelib* lub podobnym, dla którego możesz utworzyć skrót na pulpicie albo w innym miejscu tak, aby mieć do niego wygodny dostęp za pomocą jednego kliknięcia (o ile będzie Ci to potrzebne).
- W systemie *Windows 8* poszukaj środowiska IDLE pośród kafelków startowych, wyszukując słowo *idle*, przeglądając ekran startowy *Aplikacje* lub używając *Eksploratora plików*, aby znaleźć wspomniany wcześniej plik *idle.pyw*. Szczególnie w tym systemie możesz potrzebować utworzenia skrótu, ponieważ w trybie pulpu nie ma przycisku *Start* (prynajmniej dzisiaj; więcej szczegółowych informacji na ten temat znajdziesz w dodatku A).
- W systemie *macOS* wszystko, co jest wymagane do działania środowiska IDLE, jest obecne w postaci standardowych komponentów w systemie operacyjnym. IDLE powinno być dostępne do uruchomienia w aplikacjach w folderze programu *MacPython* (lub *Python N.M*). Warto tutaj wspomnieć, że niektóre wersje systemu macOS mogą wymagać zainstalowania zaktualizowanej obsługi biblioteki *tkinter*; więcej szczegółowych informacji na ten temat znajdziesz na stronie pobierania w witrynie python.org.
- W systemie *Linux* środowisko IDLE jest zwykle dostępne jako standardowy komponent. Może on mieć postać pliku wykonywalnego lub skryptu dostępnego w ścieżce systemowej; aby to sprawdzić, wpisz w wierszu poleceń powłoki polecenie *idle*. Na niektórych komputerach wymagane może być zainstalowanie dodatkowych pakietów (więcej szczegółowych informacji na ten temat znajdziesz w dodatku A), a na innych może być konieczne uruchomienie z poziomu wiersza poleceń skryptu *idle.py*, znajdującego się w podkatalogu */usr/lib/idlelib* Pythona (dokładną lokalizację możesz znaleźć za pomocą polecenia *find*).

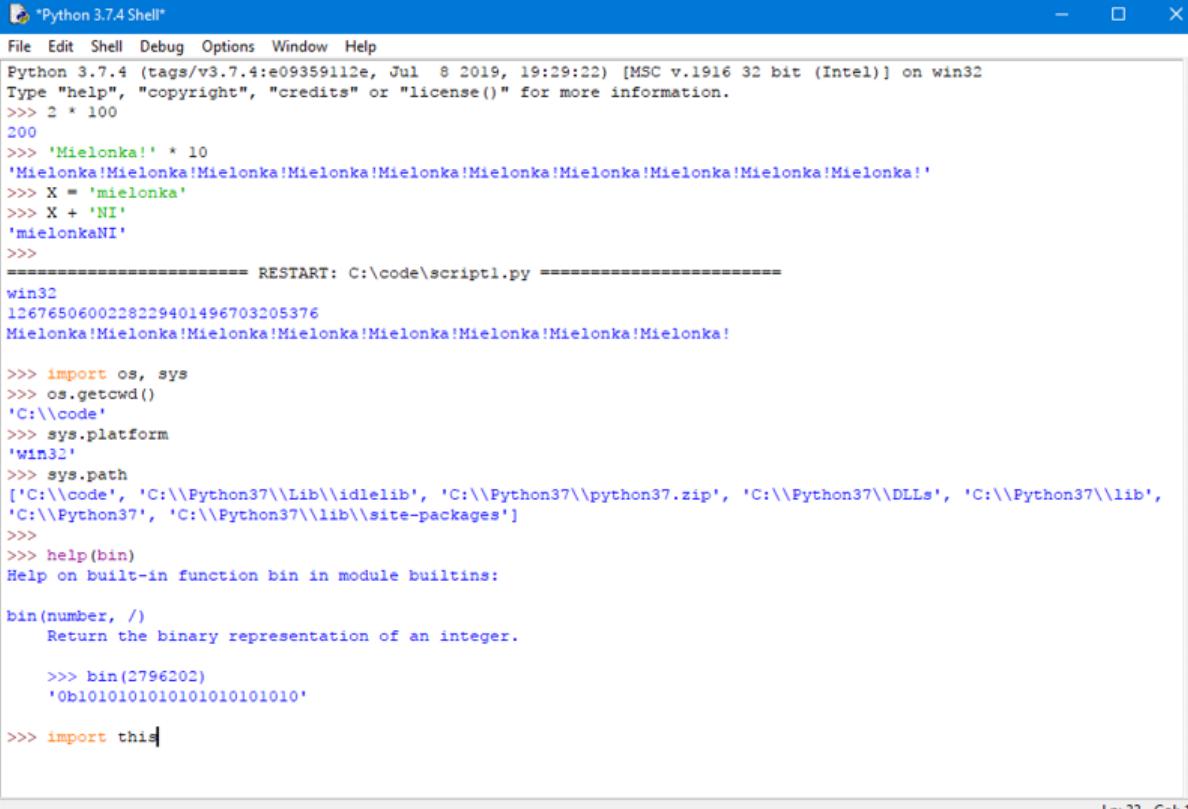
Ponieważ środowisko IDLE jest po prostu skryptem Pythona dostępnym w ścieżce wyszukiwania modułów, można je również uruchomić na dowolnej platformie i z dowolnego katalogu, wykonując z poziomu wiersza poleceń powłoki systemowej polecenie pokazane poniżej. Szczegółowe informacje na temat opcji *-m* znajdziesz w dodatku A, a w części V tej książki dowieś się więcej na temat wymaganej tutaj składni wywołania modułu z użyciem kropki (.); teraz powinieneś po prostu zaufać, że tak to właśnie powinno wyglądać:

```
c:\code> python -m idlelib.idle      # Uruchom skrypt idle.py ze ścieżki
modułu
```

Aby dowiedzieć się czegoś więcej na temat zagadnień związanych z instalacją i użytkowaniem Pythona w systemie Windows i na innych platformach, powinieneś zapoznać się zarówno z dodatkiem A, jak i sekcjami dotyczącymi instalacji i konfiguracji, znajdującymi się w dokumentacji Pythona.

Podstawy Środowiska IDLE

Przejdzmy od razu do przykładu. Na rysunku 3.3 przedstawiono wygląd okna środowiska IDLE po uruchomieniu w systemie Windows. Okno powłoki, które pojawia się na ekranie, jest głównym oknem środowiska IDLE z interaktywną sesją powłoki (zwróć uwagę na znaki zachęty *>>>*). Działa to dokładnie tak, jak wszystkie inne sesje interaktywne — wpisywany kod jest wykonywany natychmiast, dzięki czemu możemy używać tego środowiska jako narzędzia do testowania i eksperymentowania.



The screenshot shows the Python 3.7.4 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains several lines of Python code demonstrating string manipulation, imports, and the bin() function. The output area shows the results of these operations, including a long string of 'Mielonka!' repeated 200 times, the concatenation of 'mielonka' and 'NI', and the output of the bin() function for the integer 2796202. The status bar at the bottom right indicates Ln: 33 Col: 15.

```
Python 3.7.4 (tags/v3.7.4-e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 2 * 100
200
>>> 'Mielonka!' * 10
'Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!'
>>> X = 'mielonka'
>>> X + 'NI'
'mielonkaNI'
>>>
===== RESTART: C:\code\script1.py =====
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
>>> import os, sys
>>> os.getcwd()
'C:\\\\code'
>>> sys.platform
'win32'
>>> sys.path
['C:\\\\code', 'C:\\\\Python37\\\\Lib\\\\idlelib', 'C:\\\\Python37\\\\python37.zip', 'C:\\\\Python37\\\\DLLs', 'C:\\\\Python37\\\\lib', 'C:\\\\Python37', 'C:\\\\Python37\\\\lib\\\\site-packages']
>>>
>>> help(bin)
Help on built-in function bin in module builtins:

bin(number, /)
    Return the binary representation of an integer.

>>> bin(2796202)
'0b101010101010101010101010'

>>> import this
```

Rysunek 3.3. Okno główne powłoki Pythona w środowisku IDLE działającym w systemie Windows. Aby utworzyć nowy plik źródłowy (New File) lub otworzyć istniejący (Open...), możesz użyć odpowiednich poleceń z menu File. W menu Run można znaleźć polecenie Run Module, uruchamiające kod wprowadzony w tym oknie

IDLE wykorzystuje standardowe menu ze skrótami klawiaturowymi przypisanymi do większości operacji. Aby utworzyć nowy plik skryptu, użyj polecenia *File/New Window* (plik/nowe okno), czyli inaczej mówiąc, w głównym oknie środowiska IDLE wybierz z menu głównego polecenie *File*, a następnie *New Window* (począwszy od wersji 3.3.3 i 2.7.6 *New File*; nowy plik). Na ekranie pojawi się okno edytora tekstu, w którym możesz wpisywać, modyfikować, zapisywać i uruchamiać kod źródłowy nowego programu. Aby otworzyć istniejący plik do edycji lub uruchomienia, z menu głównego wybierz polecenie *File/Open* (plik/otwórz).

Choć w książce tego nie widać, środowisko IDLE *koloruje* kod programu zgodnie ze składnią — zarówno ten wpisany w oknie głównym, jak i we wszystkich oknach edycyjnych. Inny kolor mają słowa kluczowe, inny literały i tak dalej. Takie rozwiążanie znakomicie ułatwia analizowanie kodu programu, a także pozwala szybciej zauważać błędy; przykładowo ciągi znaków przypadkowo rozciągające się na kilka wierszy są wyświetlane w jednym kolorze).

Aby uruchomić plik tworzony w środowisku IDLE, powinieneś otworzyć okno edytora pliku, a następnie z menu *Run* wybrać opcję *Run Module* (lub skorzystać z odpowiedniego skrótu klawiaturowego podanego w menu). Python poinformuje Cię, że musisz najpierw zapisać swój plik, jeżeli zmieniłeś go od czasu jego otwarcia i zapomniałeś zapisać swoje zmiany — to dość często popełniany błąd, gdy nadmiernie skoncentrujesz się na programowaniu.

Kiedy uruchomisz plik programu w taki sposób, wyniki działania oraz ewentualne komunikaty o błędach są wyświetlane w głównym oknie środowiska IDLE (oknie powłoki Pythona). Na przykład, na rysunku 3.3, po wierszu ze słowem **RESTART**, mniej więcej w połowie okna, widać trzy wiersze wyników, wskazujące na wykonanie skryptu *script1.py*, otwartego w osobnym

oknie edytora. Komunikat RESTART informuje, że proces wykonywania kodu użytkownika rozpoczął się ponownie w celu wykonania edytowanego skryptu. Komunikat ten służy również do oddzielenia wyników działania skryptu (nie pojawia się, jeżeli uruchomiłeś środowisko IDLE w trybie bez podprocesu kodu użytkownika — więcej informacji na ten temat znajdziesz już za moment).

Wybrane funkcje środowiska IDLE

Podobnie jak w przypadku większości środowisk graficznych, najlepszym sposobem na poznanie środowiska IDLE jest przetestowanie go samodzielnie, choć sposoby wykonywania niektórych operacji mogą się wydawać nieco mniej oczywiste. Na przykład, jeżeli chcesz powtórzyć poprzednie polecenia w głównym oknie interaktywnej sesji środowiska IDLE, możesz używać kombinacji klawiszy *Alt+P*, aby przechodzić do poprzednich pozycji w historii poleceń, a kombinacji *Alt+N*, aby przechodzić do kolejnych poleceń (na niektórych komputerach Mac powinieneś zamiast tego użyć kombinacji klawiszy odpowiednio *Ctrl+P* i *Ctrl+N*). Twoje poprzednie polecenia zostaną przywoływane i wyświetlane w wierszu poleceń, gdzie mogą być edytowane i ponownie uruchamiane.

Möżesz także przywoływać poprzednio wykonane polecenia, klikając je lewym przyciskiem myszy i naciskając klawisz *Enter*, co spowoduje wstawienie takiego „klikniętego” polecenia do wiersza poleceń. Zamiast tego możesz również używać standardowych operacji wycinania i wklejania, chociaż obie techniki wymagają wykonania kilku kroków i czasami mogą być wykonane nieco przypadkowo. Poza środowiskiem IDLE możliwe jest przywoływanie polecenia wykonanego w sesji interaktywnej za pomocą klawiszy strzałek.

Oprócz historii poleceń i kolorowania składni, środowisko IDLE ma wiele dodatkowych, użytkowych funkcji, takich jak:

- Mechanizm *auto-indent*, czyli automatyczne tworzenie odpowiednich wcięć kodu źródłowego w edytorze (klawisz *Backspace* zmniejsza wcięcie o jeden poziom).
- Mechanizm *auto-completion*, czyli automatyczne dopełnianie słów kluczowych podczas pisania, wywoływanie przez naciśnięcie klawisza *Tab*.
- Podpowiedzi ekranowe zawierające informacje o *składni wywołania danej funkcji*, pojawiające się po wpisaniu jej nazwy i nawiasu otwierającego *(*.
- Podpowiedzi ekranowe zawierające informacje o *atributach obiektów*, pojawiające się, kiedy wpiszesz nazwę obiektu, kropkę i poczekasz przez sekundę lub naciśniesz klawisz *Tab*.

Niektóre z tych mechanizmów mogą nie działać na każdej platformie, a niektóre możesz osobno konfigurować lub wyłączać, jeżeli okaże się, że ich ustawienia domyślne po prostu przeszkadzają Ci w pracy.

Zaawansowane narzędzia środowiska IDLE

Poza podstawowymi opcjami, takimi jak edycja i uruchamianie programów, środowisko IDLE udostępnia również bardziej zaawansowane możliwości, takie jak *graficzny debugger* czy *przeglądarka obiektów*. Debugger środowiska IDLE jest włączany za pomocą menu *Debug*, natomiast przeglądarka obiektów znajduje się w menu *File*. Przeglądarka umożliwia zarówno przeglądanie klas, jak i przechodzenie przez ścieżkę wyszukiwania modułu do plików i obiektów w plikach; kliknięcie pliku lub obiektu powoduje otwarcie odpowiedniego źródła w oknie edycji tekstu.

Debugowanie programu w środowisku IDLE możesz rozpoczęć wybierając z menu w głównym oknie programu polecenie *Debug/Debugger*, a następnie uruchamiając skrypt za pomocą polecenia *Run/Run Module* w oknie edytora tekstu. Po włączeniu debugera w kodzie można ustawić punkty przerwania, klikając prawym przyciskiem myszy odpowiednie wiersze w

edytorze tekstowym; służą one do zatrzymywania wykonywania programu. W trakcie debugowania można również wyświetlać wartości zmiennych, a także obserwować działanie programu — aktualnie wykonywany wiersz kodu jest wyróżniany.

Aby ułatwić wyszukiwanie i naprawianie błędów programu, możesz kliknąć prawym przyciskiem myszy tekst komunikatu o błędzie, aby szybko przejść do wiersza kodu, w którym wystąpił błąd — jest to trik, który zdecydowanie ułatwia i przyspiesza usuwanie problemów i ponowne uruchamianie programów. Dodatkowo edytor tekstu w środowisku IDLE zawiera wiele narzędzi przydatnych dla programistów, w tym mechanizm automatycznego tworzenia wcięć w kodzie źródłowym, oraz obsługuje zaawansowane operacje wyszukiwania tekstu i plików, których nie będziemy tutaj opisywać. Ponieważ graficzny interfejs użytkownika środowiska IDLE jest dosyć intuicyjny, powinieneś po prostu samodzielnie z nim eksperymentować, aby poznać jego inne narzędzia.

Uwagi praktyczne — korzystanie ze środowiska IDLE

Środowisko IDLE jest darmowe, łatwe w użyciu, przenośne i domyślnie dostępne na większości platform. Polecam je zwłaszcza początkującym użytkownikom, ponieważ ułatwia ono wykonywanie wielu zadań i nie zakłada, że każdy ma wcześniejsze doświadczenia z korzystaniem z systemowego wiersza poleceń. W porównaniu z bardziej zaawansowanymi, komercjalnymi środowiskami programistycznymi środowisko IDLE jest jednak nieco ograniczone i niektórym użytkownikom jego używanie może wydawać się bardziej trudne niż korzystanie z wiersza poleceń. Aby pomóc Ci uniknąć niektórych często spotykanych pułapek, poniżej przedstawiamy listę najważniejszych zagadnień, o których powinieneś pamiętać zaczynając swoją przygodę z tym środowiskiem.

- **Przy zapisywaniu plików musisz w jawnym sposobie dodawać rozszerzenie .py.** Wspomniałem o tym przy okazji omawiania plików, jednak jest to częsty błąd popełniany w środowisku IDLE, w szczególności przez użytkowników systemu Windows. IDLE nie dodaje rozszerzenia .py automatycznie przy zapisywaniu plików. Musisz pamiętać o samodzielnym dopisaniu tego rozszerzenia, kiedy po raz pierwszy zapisujesz plik. Jeżeli tego nie zrobisz, będziesz w stanie uruchomić plik z systemowego wiersza poleceń czy środowiska IDLE, natomiast nie będziesz mógł tego pliku zimportować do sesji interaktywnej ani do innego modułu.
- **Skrypty powinieneś uruchamiać wybierając z menu edytora tekstu polecenie Run/Run Module, a nie poprzez interaktywne importowanie i przeładowywane modułów.** Wcześniej pokazywałem, że można wykonać plik poprzez jego interaktywne zimportowanie. Takie rozwiązanie może się jednak okazać skomplikowane, ponieważ po wprowadzeniu zmian pliki trzeba będzie ręcznie przeładowywać. Skorzystanie z polecenia *Run Module* w edytorze środowiska IDLE zawsze powoduje uruchomienie najbardziej aktualnej wersji pliku, podobnie jak to się dzieje w przypadku uruchomienia programu z poziomu systemowego wiersza poleceń. Środowisko IDLE przed uruchomieniem programu przypomina Ci również o konieczności uprzedniego zapisania pliku programu (o ile jest niezbędne). Pominięcie zapisania pliku to kolejny błąd często popełniany poza IDLE.
- **Przeładowywać trzeba jedynie moduły testowane w sposób interaktywny.** Tak jak w przypadku wiersza poleceń powłoki, polecenie *Run Module* zawsze wykonuje najbardziej aktualną wersję zarówno głównego skryptu, jak i wszelkich importowanych przez niego modułów. Z tego powodu stosowanie polecenia *Run Module* eliminuje wiele nieporozumień i problemów związanych z importowaniem. Niezbędne jest jedynie przeładowywanie modułów, które importujesz i testujesz w sposób interaktywny. Jeżeli zamiast tej opcji postanowisz skorzystać z importowania i przeładowywania, powinieneś pamiętać o skrótach klawiaturowych *Alt+P* i *Alt+N*, które pozwalają na przywołanie poprzednio wykonywanych poleceń.
- **Środowisko IDLE możesz dostosować do własnych potrzeb.** Aby zmienić czcionki tekstu i kolory w dowolnym oknie środowiska IDLE, powinieneś z menu *Options* wybrać

polecenie *Configure*. Możesz również dostosowywać skróty klawiszowe, ustawienia mechanizmu automatycznego tworzenia wcięć, ustawienia automatycznego dopełniania słów kluczowych i wiele innych; więcej szczegółowych informacji na ten temat znajdziesz w menu *Help*.

- **W środowisku IDLE nie ma opcji czyszczenia ekranu (jak dotąd).** Prośba o dodanie tej opcji powtarza się bardzo często (być może dlatego, że podobne środowiska IDE już ją posiadają), więc może zostanie ona w końcu zaimplementowana. Na razie jednak nie da się wyczyścić tekstu w oknie sesji interaktywnej. Jeżeli chcesz usunąć taki tekst z widoku, możesz albo przytrzymać klawisz *Enter*, albo uruchomić pętlę Pythona wyświetlającą serię pustych wierszy (oczywiście nikt naprawdę nie korzysta z tej ostatniej opcji, ale brzmi to znacznie bardziej technicznie niż proste naciśnięcie klawisza *Enter*!).
- **Interfejsy graficzne oparte na bibliotece tkinter oraz programy wielowątkowe mogą nie działać zbyt dobrze w środowisku IDLE.** Ponieważ środowisko IDLE jest programem napisanym w Pythonie, wykorzystującym bibliotekę tkinter, może się po prostu zawiesić, jeżeli użyjesz go do uruchamiania niektórych bardziej zaawansowanych programów wykorzystujących tę samą bibliotekę. W najnowszych wersjach środowiska IDLE, które wykonują kod użytkownika w jednym procesie, a kod samego graficznego interfejsu użytkownika w innym, nie jest to aż tak dokuczliwe, jednak niektóre programy (zwłaszcza te wykorzystujące wielowątkowość) mogą spowodować zawieszenie się interfejsu graficznego. W pewnych okolicznościach nawet proste wywołanie w kodzie programu funkcji *quit* biblioteki tkinter, czyli normalny sposób na zakończenie działania interfejsu GUI, może być wystarczające, aby po uruchomieniu w środowisku IDLE spowodować zawieszenie się Twojego programu (w takiej sytuacji lepszym rozwiązaniem będzie użycie funkcji *destroy*). Twój kod może nie sprawiać takich problemów, jednak generalna zasada brzmi, że bezpieczniej jest wykorzystywać środowisko IDLE tylko do pisania programów wyposażonych w interfejs graficzny, natomiast do uruchamiania ich lepiej jest stosować inne opcje, takie jak klikanie ikon czy używanie systemowego wiersza poleceń. Kiedy masz wątpliwości, czy Twój kod zawodzi tylko w IDLE, powinieneś go przetestować poza tym środowiskiem.
- **Jeżeli pojawiają się problemy z uruchamianiem, powinieneś spróbować uruchomić środowisko IDLE w trybie *single-process*.** Wydaje się, że ten problem w nowszych wersjach Pythona już nie występuje, ale wciąż może pojawiać się u Czytelników korzystających ze starszych wersji. Ponieważ IDLE wymaga komunikacji pomiędzy osobnymi procesami użytkownika oraz interfejsu GUI, czasami pojawia się problem z uruchomieniem na niektórych platformach (najbardziej zauważalne jest to w przypadku komputerów z systemem Windows z uwagi na zapory sieciowe blokujące połączenia). Jeżeli napotkasz taki problem, zawsze możesz spróbować uruchomić IDLE z wiersza poleceń w sposób zmuszający go do działania w trybie z jednym procesem, bez podprocesu kodu użytkownika, co pozwala uniknąć problemów z komunikacją. Tryb ten wymuszany jest przez użycie w wierszu wywołania opcji *-n*. Na przykład, w systemie Windows powinieneś uruchomić wiersz poleceń, przejść do katalogu *C:\Python3x\Lib\idlelib* i wpisać polecenie *idle.py -n* (aby przejść do tego katalogu, powinieneś użyć polecenia *cd*). Zamiast tego możesz również użyć polecenia *python -m idlelib.idle -n*, które działa z dowolnego katalogu w systemie (więcej szczegółowych informacji na temat opcji *-m* znajdziesz w dodatku A).
- **Pamiętaj o pewnych ułatwieniach dostępnych tylko w środowisku IDLE.** IDLE bardzo stara się ułatwiać życie początkującym użytkownikom, jednak pamiętaj, że niektóre z tych ułatwień nie będą miały zastosowania poza tym środowiskiem. Na przykład, IDLE wykonuje Twój skrypt w swojej własnej, interaktywnej przestrzeni nazw, dlatego zmienne wykorzystywane w kodzie automatycznie pojawiają się w sesji interaktywnej IDLE, zatem nie zawsze będziesz musiał wykonywać polecenie *import*, aby mieć dostęp do zmiennych najwyższego poziomu z plików, które już były wykonywane. Oczywiście może to być bardzo przydatne, ale także mylące, ponieważ poza środowiskiem IDLE przed użyciem nazw z innych plików trzeba je najpierw zaimportować.

Kiedy uruchamiasz plik kodu, IDLE nie tylko automatycznie przechodzi do *katalogu* takiego pliku, ale także dodaje ten katalog do ścieżki wyszukiwania importowanych modułów. Jest

to bardzo przydatna opcja pozwalająca na importowanie plików bez ustawiania ścieżki wyszukiwania, ale także coś, co nie będzie działało w ten sam sposób, kiedy będziesz uruchamiać swoje pliki poza środowiskiem IDLE. Korzystanie z takich opcji nie jest niczym niewłaściwym, jednak należy pamiętać, że są to właściwości środowiska IDLE, a nie samego Pythona.

Inne środowiska IDE

Ponieważ IDLE jest darmowe, przenośne i jest standardową częścią Pythona, jest dobrym kandydatem na pierwsze środowisko programistyczne, z którym powinieneś się zapoznać (jeżeli oczywiście w ogóle planujesz korzystać z takich rozwiązań). Osobom początkującym polecam wykonywanie ćwiczeń z książki właśnie w środowisku IDLE, o ile nie są jeszcze zaznajomione z trybem wykorzystywania wiersza polecen i preferują takie rozwiązanie. Istnieje jednak wiele alternatywnych środowisk programistycznych; niektóre z nich są znacznie bardziej rozbudowane od IDLE i mają więcej możliwości. Poniżej znajduje się lista kilku najpopularniejszych.

Eclipse i PyDev

Eclipse to zaawansowane środowisko programistyczne na licencji open source, wyposażone w graficzny interfejs użytkownika. Powstało jako środowisko IDE dla języka Java, jednak po zainstalowaniu dodatku PyDev (lub innego dodatku o podobnych funkcjach) obsługuje również programowanie w Pythonie. Eclipse jest popularnym wyborem dla programowania w Pythonie, a możliwości tego programu znacznie wykraczają poza opcje dostępne w środowisku IDLE. Obejmują one między innymi uzupełnianie kodu, kolorowanie składni, analizę składni, refaktoryzację kodu oraz debugowanie. Wadą środowiska Eclipse są jego pokaźne rozmiary, a pewne możliwości mogą wymagać konieczności instalowania rozszerzeń na licencji *shareware* (z czasem może się to zmienić). Jeżeli jednak czujesz, że czas przejść z IDLE na coś bardziej zaawansowanego, to zdecydowanie połączenie Eclipse i PyDev powinno być warte Twojej uwagi.

Komodo

Komodo to zaawansowane środowisko programistyczne z interfejsem GUI, przeznaczone dla Pythona i innych języków programowania. Zawiera opcje kolorowania składni, edycji tekstu, debugowania, a także wiele innych. Komodo oferuje również wiele możliwości, które nie są dostępne w środowisku IDLE, w tym tworzenie projektów, integrację z systemem kontroli kodu źródłowego, debugowanie wyrażeń regularnych i inne. W tej chwili środowisko Komodo nie jest darmowe; można je pobrać ze strony <http://www.activestate.com>. Firma ActiveState oferuje również dystrybucję ActivePython, o której powiemy kilka słów w dodatku A.

NetBeans IDE for Python

NetBeans to graficzne środowisko programistyczne na licencji open source udostępniające wiele zaawansowanych opcji przydatnych programistom Pythona — między innymi uzupełnianie kodu, automatyczne tworzenie wcięć w kodzie, kolorowanie składni, podpowiedzi ekranowe w edytorze kodu, składanie kodu, refaktoryzację, debugowanie, testowanie kodu oraz tworzenie projektów. Za jego pomocą można programować zarówno w CPythonie, jak i w Jythonie. Tak jak w przypadku Eclipse, instalowanie NetBeans wymaga dodatkowych kroków wykraczających poza to, co jest potrzebne dla IDLE, jednak wiele osób uważa, że jego możliwości są tego warte. Więcej szczegółowych informacji na ten temat znajdziesz na stronie projektu NetBeans.

PythonWin

PythonWin jest darmowym środowiskiem programistycznym dla Pythona przeznaczonym dla systemu Windows i udostępnianym jako część dystrybucji ActivePython firmy ActiveState (program ten można również pobrać oddzielnie z zasobów umieszczonych w witrynie <http://www.python.org/>). Przypomina ono środowisko IDLE; dodano do niego również kilka przydatnych rozszerzeń dla systemu Windows — na przykład obsługę obiektów COM. Aktualnie IDLE jest chyba bardziej zaawansowanym środowiskiem niż PythonWin (na przykład dwuprocesowa architektura IDLE pozwala łatwiej zapobiegać zawieszaniu się systemu). PythonWin oferuje jednak narzędzia dla programowania w systemie Windows, które w IDLE nie są dostępne.Więcej informacji na temat tego środowiska można znaleźć na stronie <http://www.activestate.com>.

Wing, Visual Studio i inne

Inne środowiska IDE są również popularne wśród programistów Pythona, w tym głównie komercyjne *IDE Wing*, *Microsoft Visual Studio* (za pośrednictwem odpowiedniej wtyczki), oraz *PyCharm*, *PyScripter*, *Pyshield* i *Spyder* — ale nie mam tutaj miejsca, aby je choćby pokrótko opisać, a z czasem z pewnością pojawi się jeszcze więcej takich środowisk. W praktyce niemal każdy *edytor tekstu* przeznaczony dla programistów obsługuje obecnie programowanie w Pythonie (przynajmniej w jakimś stopniu); czasem obsługa Pythona jest od razu wbudowana, a innym razem trzeba ją zainstalować osobno. Przykładowo, edytory *Emacs* i *Vim* całkiem nieźle radzą sobie z pisaniem programów w Pythonie.

Wybór środowiska IDE jest często bardzo subiektywny, więc zachęcam do szukania narzędzi, które pasują do Twojego stylu programowania i zadań, jakie realizujesz.Więcej szczegółowych informacji na temat zintegrowanych środowisk programistycznych dla Pythona możesz znaleźć na stronie <http://www.python.org> lub poszukać w sieci, wpisując w oknie wyszukiwarki frazę „Python IDE” lub coś podobnego. Z kolei wyszukiwanie frazy „Python editors” zwraca m.in. łącze do strony wiki, na której znajdują się informacje o wielu środowiskach programistycznych oraz edytorach tekstu przeznaczonych programistów tworzących swoje projekty w języku Python.

Inne opcje wykonywania kodu

Dowiedziałeś się już, jak uruchamiać kod wpisywany interaktywnie i jak na wiele sposobów uruchamiać kod zapisany w plikach — korzystając z systemowego wiersza poleceń, poprzez klikanie ikon plików, wykonywanie poleceń `import` i funkcji `exec` czy korzystanie z graficznych środowisk programistycznych, takich jak IDLE. To większość powszechnie używanych technik i w zupełności wystarczą Ci one do uruchomienia zdecydowanej większości przykładów omawianych w tej książce. Istnieją jednak jeszcze inne sposoby uruchamiania kodu napisanego w Pythonie, z których większość ma specjalne lub zawiżone przeznaczenie. Poniżej przyjrzymy się kilku z nich.

Osadzanie wywołań

W pewnych specyficznych zastosowaniach kod Pythona może również być automatycznie wykonywany przez system go zawierający. W takich przypadkach mówimy, że kod w języku Python jest *osadzony* (i wykonywany) w innym programie. Sam kod w Pythonie może na przykład zostać umieszczony w pliku tekstowym, przechowywany w bazie danych, pobrany ze strony HTML czy odczytany z dokumentu XML, ale z operacyjnego punktu widzenia to inny system, a nie Ty, w odpowiednim momencie nakazuje uruchomienie kodu, który utworzyłeś.

Taki osadzony tryb wykonywania jest często wykorzystywany w dostosowywaniu programów do potrzeb użytkowników końcowych. Na przykład, w wielu popularnych dzisiaj grach dozwolone jest dokonywanie modyfikacji za pomocą dostępnego dla użytkownika osadzonego kodu

napisanego w Pythonie, który wykonywany jest w odpowiednich momentach rozgrywki. Użytkownicy mogą modyfikować działanie wybranych elementów gry, tworząc i udostępniając kod w języku Python. Ponieważ kod taki jest interpretowany, nie ma konieczności ponownej komplikacji całej aplikacji w celu uwzględnienia zmian (więcej szczegółowych informacji na temat uruchamiania kodu w Pythonie znajdziesz w rozdziale 2.).

W takim scenariuszu system, w którym osadzany jest kod Pythona, może być napisany w językach C, C++ czy nawet Java (jeżeli korzystamy z Jythona). Można na przykład tworzyć i uruchamiać fragmenty kodu w Pythonie z poziomu programu w języku C, wywołując odpowiednie funkcje interfejsu runtime API Pythona:

```
#include <Python.h>
...
Py_Initialize();                                     // To kod w języku C,
PyRun_SimpleString("x = 'waleczny ' + 'sir robin'"); // ale wykonuje kod w
                                                       Pythonie
```

W tym fragmencie kodu program napisany w języku C osadza interpreter Pythona, dodając jego biblioteki, i przekazuje do niego instrukcję przypisania, którą należy wykonać. Programy napisane w C mogą również uzyskać dostęp do modułów oraz obiektów Pythona i przetwarzać lub wykonywać je z użyciem innych narzędzi API Pythona.

Niniejsza książka nie jest poświęcona integracji kodu napisanego w C i Pythonie, jednak warto pamiętać, że w zależności od planowanych zastosowań, programy w Pythonie nie zawsze muszą być uruchamiane z poziomu Pythona. Nie zmienia to jednak w niczym faktu, że nadal możesz wykorzystywać omówione wcześniej techniki uruchamiania programów do testowania kodu niezależnie od systemu, w którym zostanie docelowo osadzony^[6].

Zamrożone binarne pliki wykonywalne

Zamrożone binarne pliki wykonywalne, opisane w rozdziale 2., to pakiety łączące kod bajtowy programu z interpreterem Pythona w jeden wykonywalny program. Dzięki temu programy napisane w tym języku mogą być uruchamiane w taki sam sposób jak inne programy wykonywalne (czyli na przykład przez kliknięcie ikony czy wiersz poleceń). Choć ta opcja sprawdza się w przypadku dostarczania gotowego produktu, właściwie nie jest przeznaczona do użycia w trakcie tworzenia programu. Zazwyczaj kod zamraża się tuż przed udostępnieniem go, już po zakończeniu fazy programowania. Więcej informacji na temat tej opcji znajdziesz w poprzednim rozdziale.

Uruchamianie kodu z poziomu edytora tekstu

Jak wspomniano wcześniej, większość edytorów tekstu przeznaczonych dla programistów — nawet jeśli nie spełniają wymagań pełnego środowiska programistycznego — umożliwia jednak edycję i być może wykonywanie programów w Pythonie. Opcja ta może być albo od razu wbudowana, albo łatwo rozszerzalna za pomocą dodatków dostępnych w internecie. Osoby zaznajomione z edytorem Emacs mogą na przykład edytować i uruchamiać kod napisany w Pythonie bezpośrednio w tym programie. Więcej informacji na ten temat znajduje się na stronie <http://www.python.org/editors>; warto również poszukać tych informacji w internecie, na przykład wpisując „Python editors” w wyszukiwarce.

Jeszcze inne możliwości uruchamiania

W zależności od wykorzystywanej platformy dostępne mogą być inne sposoby uruchamiania programów napisanych w Pythonie. Niektóre systemy Macintosh pozwalają na przykład na przeciąganie ikony plików programów na ikonę interpretera Pythona. W systemie Windows zawsze można uruchomić skrypt w Pythonie za pomocą opcji *Uruchom...* z menu *Start*. Dodatkowo biblioteka standardowa Pythona zawiera również narzędzia, które pozwalają na uruchamianie w osobnych procesach programów napisanych w Pythonie za pomocą innych programów w tym języku (na przykład `os.popen`, `os.system`). Skrypty napisane w Pythonie można również uruchamiać w większych kontekstach, na przykład w interenie (skrypt może zostać uruchomiony na serwerze za pośrednictwem strony internetowej). Takie rozwiązania wykraczają jednak daleko poza zakres niniejszego rozdziału.

Przyszłe możliwości?

Choć niniejszy rozdział odzwierciedla stosowane obecnie praktyki, duża jego część jest ograniczona do określonej platformy i czasu. Wiele zaprezentowanych tutaj możliwości uruchamiania i wykonywania kodu pojawiło się już po napisaniu poprzednich wydań tej książki. Tak jak w przypadku możliwości wykonywania programów, niewykluczone jest, że z czasem pojawią się nowe opcje ich uruchamiania.

Również nowe systemy operacyjne czy nowe wersje istniejących systemów mogą udostępnić rozwiązania wykraczające poza opisane w tym rozdziale. Ponieważ Python na ogół dostosowuje się do takich zmian, programy napisane w tym języku powinno się dać uruchomić w dowolny sposób, jakiego używają wykorzystywane przez nas urządzenia — zarówno teraz, jak i w przyszłości. Nie ma znaczenia, czy będzie to rysowanie na tablecie z ekranem dotykowym albo na smartfonie, przeciąganie ikon w rzeczywistości wirtualnej czy wykrykiwanie nazwy skryptu nad głowami naszych współpracowników.

Zmiany implementacyjne mogą również wpływać w pewnym stopniu na schematy uruchamiania (na przykład pełny kompilator może wytwarzać normalne pliki wykonywalne, które uruchamiane będą podobnie do dzisiejszych zamrożonych binarnych plików wykonywalnych). Gdybym jednak wiedział, co przyniesie nam przyszłość, prawdopodobnie rozmawiałbym teraz z jakimś maklerem giełдовym, a nie pisał te słowa!

Jaką opcję wybrać?

Skoro mamy tyle możliwości, pojawia się pytanie: „Jaka opcja będzie dla mnie najlepsza?”. Generalnie osobom zaczynającym przygodę z programowaniem w Pythonie polecam interfejs IDLE. Zawiera on przyjazny graficzny interfejs użytkownika i jest w stanie ukryć pewne szczegółowe konfiguracyjne. IDLE zawiera również niezależny od platformy edytor tekstu służący do tworzenia skryptów, a także jest standardową i darmową częścią Pythona.

Bardziej zaawansowani programiści mogą czuć się lepiej z wybranym zwykłym edytorem tekstem w jednym oknie i uruchamianiem programów za pomocą wiersza poleceń lub klikania ikon w drugim oknie (tak właśnie swoje programy pisze autor niniejszej książki, choć należy zaznaczyć, że w przeszłości dużo korzystał z Uniksa). Ponieważ wybór środowiska programistycznego jest sprawą subiektywną, nie mogę tu zaoferować jakichś uniwersalnych reguł. Na ogół najlepszym wyborem będzie ta opcja, która jest dla nas najwygodniejsza.

Debugowanie kodu w Pythonie

Oczywiście żaden z moich czytelników ani studentów nie popełnia błędów w kodzie (*tu wstaw odpowiedniego emotikona*), ale na potrzeby naszych bardziej pechowych przyjaciół i znajomych, którym się to jednak czasem zdarza, przyjrzymy się strategiom stosowanym przez prawdziwych programistów Pythona podczas debugowania kodu:

- **Nie robimy nic.** Pisząc to, nie mam na myśli, że programiści Pythona nie debugują swojego kodu. Kiedy jednak w programie napisanym w tym języku popełnimy błąd, otrzymujemy od razu bardzo przydatny i czytelny komunikat o błędzie (kto go jeszcze nie widział, z pewnością niedługo zobaczy). Osobom znającym Pythona, zwłaszcza na potrzeby własnego kodu, często to wystarcza — dość przeczytać komunikat o błędzie i naprawić podany wiersz w podanym pliku. Dla wielu osób to właśnie oznacza „debugowanie w Pythonie”. Może to jednak nie być idealne rozwiązanie w przypadku dużych systemów, których nie napisaliśmy samodzielnie.
- **Wstawiamy instrukcję print.** Chyba najczęściej wykorzystywany przez programistów Pythona sposobem debugowania kodu (za pomocą tej metody debuguję kod i ja) jest wstawianie instrukcji `print` i ponowne wykonywanie kodu. Ponieważ kod w Pythonie wykonywany jest natychmiast po wprowadzeniu zmian, zazwyczaj jest to najszybszy sposób, aby uzyskać więcej informacji niż za pomocą samych komunikatów o błędach. Instrukcje `print` nie muszą być szczególnie wyszukane — proste „Jestem tutaj” albo wyświetlenie wartości zmiennych często daje nam wystarczający kontekst. Należy jedynie pamiętać o usunięciu lub zakomentowaniu (czyli umieszczeniu z przodu znaku `#`) dodanych instrukcji `print` przed opublikowaniem kodu!
- **Używamy debugera z graficznego interfejsu użytkownika IDE.** W przypadku większych systemów, których nie jesteśmy autorami, a także na potrzeby osób początkujących, które chcą bardziej szczegółowo śledzić kod, większość środowisk programistycznych dla Pythona zawiera jakiś rodzaj obsługi debugowania dostępnego za pomocą kliknięcia myszą. Także środowisko IDLE zawiera debugger, jednak w praktyce nie wydaje się on używany zbyt często — być może dlatego, że nie ma wiersza poleceń, a może przez to, że wstawienie do kodu instrukcji `print` jest szybsze od konfigurowania sesji debugowania w GUI. Aby dowiedzieć się więcej na ten temat, warto zajrzeć do systemu pomocy środowiska IDLE albo po prostu spróbować samemu. Podstawowy interfejs debugera opisany jest w podrozdziale „Zaawansowane opcje środowiska IDLE” w tym rozdziale. Pozostałe środowiska programistyczne, takie jak Eclipse, NetBeans, Komodo czy Wing IDE, także oferują zaawansowane debugery obsługiwane za pomocą kliknięcia myszą.Więcej informacji na ich temat można znaleźć w dokumentacji.
- **Używamy debugera pdb, działającego z poziomu wiersza poleceń.** Aby ułatwić kontrolę nad kodem, Python posiada debugger kodu źródłowego o nazwie `pdb`, będący modelem biblioteki standardowej tego języka. W `pdb` polecenia wpisuje się wiersz po wierszu, wyświetla zmienne, ustawia i kasuje punkty kontrolne czy pozwala kontynuować kod do punktu wstrzymania lub błędu. Debugger `pdb` można uruchomić interaktywnie za pomocą importowania, a także jako skrypt najwyższego poziomu. Bez względu na sposób uruchomienia, z uwagi na to, że polecenia wpisuje się w celu kontrolowania sesji, jest to narzędzie o dużych możliwościach. Zawiera ono także funkcję postmortem (`pdb.pm()`), którą można wykonać po wystąpieniu wyjątku w celu otrzymania informacji z chwili wystąpienia błędu.Więcej informacji na temat debugera `pdb` znajdziesz w dokumentacji Pythona oraz w rozdziale 36., a dodatku A znajdziesz przykłady uruchamiania debugera `pdb` z wiersza poleceń za pomocą opcji `-m` polecenia `python`.
- **Użyj argumentu -i w wierszu wywołania Pythona.** Jeżeli nie korzystasz z dodatkowych poleceń `print` ani nie używasz debugera `pdb`, to i tak nadal możesz zobaczyć, co poszło nie tak. Jeżeli uruchamiasz skrypt z wiersza poleceń i między poleceniem `python` a nazwą skryptu przekazujesz argument `-i` (np. `python -i m.py`), po zakończeniu działania skryptu Python wejdzie w interaktywny tryb interpretera (wyświetlając znak zachęty `>>>`), bez względu na to, czy jego działanie zakończyło się pomyślnie, czy też wystąpił błąd. Dzięki temu po zakończeniu działania skryptu możesz wyświetlić końcowe wartości zmiennych i uzyskać więcej szczegółowych informacji o tym, co działało się w kodzie, ponieważ zmienne znajdują się w przestrzeni nazw. Możesz także zimportować i uruchomić debugger `pdb`, aby uzyskać jeszcze więcej informacji; jego tryb postmortem pozwoli Ci również sprawdzić ostatni błąd, jeżeli działanie skryptu zakończyło się niepowodzeniem.Więcej szczegółowych informacji na temat opcji `-i` znajdziesz w dodatku A.

- **Inne opcje.** W przypadku specyficznych wymagań w zakresie debugowania kodu dodatkowe narzędzia można znaleźć wśród programów na licencji open source — w tym obsługę programów wielowątkowych, osadzonego kodu czy dołączania procesów. Przykładowo pakiet *Winpdb* to samodzielny debugger z zaawansowaną obsługą debugowania i działającym na różnych platformach interfejsem GUI oraz konsolą.

Powyższe opcje zyskają na znaczeniu, kiedy zaczniesz pisać większe skrypty. Chyba najlepszą wiadomością dotyczącą debugowania jest jednak to, że błędy są w Pythonie wykrywane i zgłaszane, a nie po cichu ignorowane czy powodujące zakończenie działania całego systemu. Tak naprawdę same błędy są dobrze zdefiniowanym mechanizmem znanym jako *wyjątki*, które można przechwytywać i przetwarzać (więcej informacji na ten temat znajduje się w siódmej części książki). Popełnianie błędów nigdy nie jest oczywiście przyjemnością, jednak mówiąc z pozycji osoby, która pamięta czasy, gdy debugowanie oznaczało wyciągnięcie kalkulatora szesnastkowego i przechodzenie w pocie czoła przez stosy wydruków zrzutów z pamięci, obsługę debugowania w Pythonie uznaję za element sprawiający, że błędy stały się o wiele mniej dotkliwe, niż mogłyby być.

Podsumowanie rozdziału

W tym rozdziale przyjrzaliśmy się popularnym sposobom uruchamiania programów napisanych w Pythonie — wykonywaniu kodu wpisanego w sposób interaktywny, wykonywaniu kodu przechowywanego w plikach za pomocą systemowego wiersza poleceń, klikaniu ikon plików, importowaniu modułów, wywoływaniu funkcji exec, a także środowiskiem programistycznym z graficznym interfejsem użytkownika, takim jak IDLE. Celem tego rozdziału było dostarczenie użytkownikowi odpowiedniej ilości informacji, która wystarczy mu do rozpoczęcia pisania kodu. Zrobimy to już w kolejnej części książki, gdzie zaczniemy omawiać sam język, rozpoczynając od podstawowych *typów danych*.

Najpierw jednak pora na tradycyjny quiz końcowy, który sprawdzi wiadomości omówione w niniejszym rozdziale. Ponieważ jest to również ostatni rozdział tej części książki, po quizie znajdziesz zbiór bardziej rozbudowanych ćwiczeń, które sprawdzą Twoją znajomość zagadnień poruszonych w całej części. Pomoc przy rozwiązywaniu ćwiczeń znajdziesz w dodatku D, który możesz również wykorzystać do odświeżenia wiedzy przedstawionej w książce. Zajrzyj do niego po wykonaniu ćwiczeń.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób można rozpocząć interaktywną sesję interpretera?
2. Gdzie powinieneś wpisać polecenie pozwalające na uruchomienie pliku skryptu?
3. Wymień cztery lub większą liczbę sposobów wykonania kodu zapisanego w pliku skryptu.
4. Wymień dwie pułapki związane z klikaniem ikon plików w systemie Windows.
5. Dlaczego czasami konieczne może być przeładowanie modułu?
6. W jaki sposób można w środowisku IDLE uruchomić skrypt?
7. Wymień dwie pułapki związane z wykorzystywaniem środowiska IDLE.

8. Co to jest przestrzeń nazw i jaki ma ona związek z plikami modułów?

Sprawdź swoją wiedzę – odpowiedzi

1. W systemie Windows 7 i wcześniejszych sesję interaktywną można rozpoczęć, klikając przycisk *Start*, a następnie wybierając polecenie *Wszystkie programy/Python/Python (command line)*. Ten sam efekt można uzyskać w systemie Windows i na innych platformach, wpisując w wierszu poleceń powłoki polecenie `python`. Alternatywnym rozwiązańiem jest uruchomienie środowiska IDLE, ponieważ jego główne okno to po prostu interaktywna sesja powłoki Pythona. W zależności od platformy i wersji Pythona, jeżeli nie dodałeś ścieżki Pythona do zmiennej systemowej PATH, być może będziesz musiał najpierw za pomocą polecenia `cd` przejść do katalogu, w którym zainstalowany jest Python, lub w wywoływanym poleceniu podać pełną ścieżkę (w systemie Windows będzie to na przykład `C:\Python3x\python`; chyba że używasz programu uruchamiającego z wersji 3.3 lub wyższej).
2. Polecenie uruchamiające skrypt można wpisać w dowolnym wierszu poleceń powłoki systemowej używanej platformy — w systemie Windows będzie to *Wiersz poleceń*, a w systemach Unix, Linux oraz macOS konsola *xterm* czy okno terminala. Pamiętaj, że polecenie takie powinieneś uruchomić z poziomu wiersza poleceń powłoki systemu, a nie interaktywnej sesji powłoki Pythona (gdzie wyświetlany jest znak zachęty w postaci `>>>`) — uważaj, aby ich nie pomylić.
3. Kod programu (skrypt) zapisany w pliku (czyli inaczej mówiąc, w module) można uruchomić z poziomu systemowego wiersza poleceń, za pomocą kliknięcia ikony pliku, importowania i przeładowywania, wbudowanej funkcji `exec`, a także wybierając odpowiednie polecenie z menu środowiska IDE; na przykład w środowisku IDLE będzie to polecenie *Run/Run Module*. W systemie Unix skrypt można także uruchomić jako plik wykonywalny za pomocą sztuczki z wierszem `#!`, a niektóre platformy obsługują jeszcze inne, bardziej wyspecjalizowane techniki uruchamiania, takie jak przeciąganie i upuszczanie. Dodatkowo niektóre edytory tekstu oferują unikalne sposoby wykonywania kodu napisanego w Pythonie, a niektóre programy napisane w Pythonie udostępniane są jako samodzielne, wykonywalne „zamrożone pliki binarne”. Niektóre systemy wykorzystują osadzony kod napisany w Pythonie, gdzie jest on wykonywany automatycznie przez programy napisane w języku takim, jak C, C++ czy Java. Ta ostatnia technika służy zazwyczaj do udostępnienia użytkownikowi warstwy pozwalającej na dostosowanie głównej aplikacji do własnych potrzeb.
4. Skrypty, które wyświetlają wyniki i kończą swoje działanie sprawiają, że okno programu natychmiast zniką, zanim jeszcze zdążymy cokolwiek zobaczyć (dlatego w takich sytuacjach przydaje się sztuczkę z wywołaniem funkcji `input`). Komunikaty o błędach wygenerowane przez skrypt również pojawiają się w oknie, które często zniką, zanim zdążymy przejrzeć jego zawartość (między innymi dlatego lepszym rozwiązaniem najczęściej okazuje się korzystanie z systemowego wiersza poleceń lub środowiska programistycznego, takiego jak IDLE).
5. Python domyślnie importuje (ładuje) moduł tylko raz na dany proces, dlatego jeżeli zmienimy jego kod źródłowy i chcemy wykonać nową wersję bez zatrzymywania i ponownego uruchamiania Pythona, trzeba go będzie przeładować. Przed przeładowaniem modułu trzeba go przynajmniej raz zaimportować. Wykonywanie plików programów z poziomu wiersza poleceń powłoki systemu, za pomocą kliknięcia ikony lub poprzez środowisko programistyczne, takie jak IDLE, sprawia,

że problem ten znika, ponieważ te sposoby za każdym razem wykonują bieżącą wersję kodu źródłowego pliku.

6. W oknie edytora tekstu powinieneś wybrać z menu *Run* opcję *Run Module*. Wykonuje ona kod źródłowy z tego okna jako plik skryptu najwyższego poziomu i wyświetla wyniki jego działania w interaktywnym oknie powłoki Pythona.
7. Pewne typy programów nadal mogą zawiesić środowisko IDLE — w szczególności dotyczy to wielowątkowych programów wyposażonych w graficzne interfejsy użytkownika (zagadnienie wielowątkowości pozostaje poza zakresem niniejszej książki). IDLE ma również pewne cechy, które mogą powodować problemy z działaniem programów poza tym środowiskiem, na przykład to, że zmienne skryptu są automatycznie importowane do przestrzeni nazw sesji interaktywnej, co jest charakterystyczne tylko dla środowiska IDLE, a nie dla samego Pythona.
8. Przestrzeń nazw to pakiet zmiennych (np. nazw). Przybiera ona w Pythonie formę obiektu wraz z atrybutami. Każdy plik modułu automatycznie staje się przestrzenią nazw, czyli pakietem zmiennych odzwierciedlającym przypisania wykonane na najwyższym poziomie pliku. Przestrzenie nazw pomagają zapobiegać konfliktom nazw w programach napisanych w Pythonie — ponieważ każdy plik modułu jest samodzielną przestrzenią nazw, pliki muszą jawnie importować inne pliki w celu użycia zmiennych z ich przestrzeni nazw.

Sprawdź swoją wiedzę — ćwiczenia do części pierwszej

Czas zabrać się za samodzielne tworzenie kodu. Pierwsza sesja z ćwiczeniami jest dość prosta, ale ma na celu upewnienie się, że jesteś gotowy do pracy z resztą książki, a kilka z jej pytań wskazuje na tematy, które będą się pojawiać w kolejnych rozdziałach. Odpowiedzi na poniższe pytania znajdują się w podrozdziale „Część I — Wprowadzenie” w dodatku D. Ćwiczenia oraz ich rozwiązania czasami zawierają informacje dodatkowe, które nie są omówione w tekście głównym książki, dlatego warto zajrzeć do odpowiedzi nawet wtedy, gdy na wszystkie pytania odpowiedziałeś samodzielnie.

1. *Interakcja.* Wykorzystując systemowy wiersz poleceń, środowisko IDLE lub inną metodę, uruchom interaktywny wiersz poleceń Pythona (ze znakiem zachęty `>>>`) i wpisz wyrażenie `"Witaj, świecie!"` wraz z cudzysłowem. Python powinien zwrócić ten sam ciąg znaków. Celem tego ćwiczenia jest skonfigurowanie naszego środowiska w taki sposób, aby można było używać Pythona. W niektórych systemach być może najpierw trzeba będzie wykonać odpowiednie polecenie `cd`, wpisać pełną ścieżkę do pliku wykonywalnego Pythona czy dodać ścieżkę Pythona do zmiennej środowiskowej `PATH`. W razie potrzeby zmienną `PATH` można ustawić w pliku `.chrc` lub `.kshrc`, tak by Python był zawsze dostępny w systemach opartych na Uniksie. W systemie Windows najlepszym miejscem do ustawienia ścieżki będzie zmienna `PATH`. Więcej szczegółowych informacji na temat ustawiania zmiennych środowiskowych znajdziesz w dodatku A.
2. *Programy.* W wybranym edytorze tekstu utwórz prosty plik modułu zawierający pojedynczą instrukcję `print('Witaj, module!')` i zapisz go jako `module1.py`. Po zapisaniu powinieneś uruchomić ten plik w dowolny sposób — w środowisku IDLE, klikając ikonę pliku, przekazując jako argument wywołania interpretera Pythona w systemowym wierszu poleceń powłoki (na przykład `python module1.py`), wywołując wbudowaną funkcję `exec` czy importując i przeładowując moduł. Najlepiej będzie,

jeżeli poeksperimentujesz z uruchamianiem pliku za pomocą maksymalnie dużej liczby technik zaprezentowanych w niniejszym rozdziale. Które techniki wydają się najłatwiejsze? (Na to pytanie nie ma oczywiście jednoznacznej odpowiedzi).

3. *Moduły.* Uruchom interaktywną sesję Pythona (ze znakiem zachęty `>>>`) i zimportuj moduł napisany w ćwiczeniu 2. Przenieś plik do innego katalogu i spróbuj jeszcze raz zimportować go z poprzedniego, oryginalnego katalogu (np. uruchom Pythona oryginalnym katalogu podczas importowania). Co się stanie? (Wskazówka: czy w oryginalnym katalogu jest jeszcze plik kodu bajtowego `module1.pyc` lub podobny plik w podkatalogu `_pycache_`?).
4. *Skrypty.* Jeżeli Twoja platforma obsługuje tę opcję, dodaj na początku pliku modułu `module1.py` wiersz ze znakami `#!`, nadaj plikowi prawa do wykonywania i uruchom go bezpośrednio jako plik wykonywalny. Co musi zawierać pierwszy wiersz? Znaki `#!` zazwyczaj mają znaczenie tylko na platformach Unix, Linux i w systemach podobnych do Uniksa, takich jak macOS. Osoby pracujące w systemie Windows mogą zamiast tego spróbować uruchomić plik, podając samą jego nazwę w oknie wiersza poleceń bez poprzedzającego ją słowa `python` (taka opcja zadziała w nowszych wersjach tego systemu) lub za pomocą okna dialogowego *Uruchom...* z menu *Start*. Jeżeli używasz Pythona 3.3 i nowszych lub instalowanego wraz z nim programu uruchamiającego, poeksperimentuj ze zmianą wiersza `#!` tak, aby uruchamiać różne wersje Pythona, które możesz zainstalować na swoim komputerze (więcej szczegółowych informacji na ten temat znajdziesz w dodatku B).
5. *Błędy i debugowanie.* Poeksperimentuj z wpisywaniem wyrażeń matematycznych i instrukcji przypisania w interaktywnym wierszu poleceń Pythona. Przy okazji spróbuj wpisać wyrażenia takie jak `2 ** 500` czy `1 / 0`, a także odwołaj się do niezdefiniowanej nazwy zmiennej, tak jak zrobiliśmy wcześniej w tym rozdziale. Co się stanie?

Möżesz jeszcze nie zdawać sobie sprawy z tego, że kiedy robisz błędy, przetwarzasz wyjątki (zagadnienie to zostanie omówione szerzej w części siódmej). Jak się okazuje, w takiej sytuacji wywołujesz coś, co znane jest pod nazwą *domyślnej procedury obsługi wyjątków* (ang. *default exception handler*), która wyświetla na ekranie standardowe komunikaty o błędach. Jeżeli dany błąd nie będzie obsługiwany przez Twój program, zostanie przechwycony przez domyślną procedurę obsługi wyjątków, która wyświetli na ekranie standardowy komunikat o wystąpieniu błędu.

Wyjątki są w Pythonie powiązane z pojęciem *debugowania*. Jeżeli dopiero zaczynasz programowanie w tym języku, domyślne komunikaty o błędach najprawdopodobniej Ci wystarczą — podają one przyczynę błędu, a także wskazują wiersze kodu gdzie wystąpił błąd.Więcej informacji na temat debugowania znajduje się w ramce „Debugowanie kodu w Pythonie” w tym rozdziale.

6. *Przerwania i cykle.* W wierszu poleceń Pythona wpisz poniższy kod:

```
L = [1, 2]                                     # Utworzenie listy
dwuelementowej

L.append(L)                                    # Dodanie listy L jako
elementu do samej siebie

L                                         # Wyświetlenie listy L
```

Co się dzieje? We wszystkich nowszych wersjach Pythona można zobaczyć dziwnie wyglądające wyniki działania, które omówimy w dodatku z rozwiązaniami do ćwiczeń, a które okażą się mieć większy sens po omówieniu referencji w dalszej części książki. Jeżeli używasz Pythona starszego niż wersja 1.5.1, to na większości platform najprawdopodobniej pomoże Ci tutaj naciśnięcie kombinacji klawiszy `Ctrl+C`. Dlaczego Twoja wersja Pythona reaguje w taki sposób na powyższy kod?



Jeżeli rzeczywiście używasz Pythona starszego niż wersja 1.5.1 (miejmy nadzieję, że zdarza się to już bardzo rzadko!) powinieneś przed wykonaniem tego testu upewnić się, że Twój komputer jest w stanie zatrzymać program za pomocą jakiegoś rodzaju kombinacji klawiszy przerywającej jego działanie. W przeciwnym razie musisz być przygotowany na naprawdę długie oczekiwanie.

7. *Dokumentacja*. Spróbuj poświęcić przynajmniej kwadrans na zapoznanie się z bibliotekami Pythona oraz dokumentacją i sprawdzenie, jakie narzędzia dostępne są w tej bibliotece oraz jaka jest struktura zbioru dokumentacji. Aby zapoznać się z lokalizacją najważniejszych zagadnień w dokumentacji, potrzebne Ci będzie co najmniej tyle czasu; kiedy będziesz to miał już za sobą, o wiele łatwiej będzie Ci znaleźć to, czego potrzebujesz. Dokumentację można znaleźć za pomocą polecenia *Python* z menu *Start* (w przypadku systemu Windows), za pomocą polecenia *Python Docs* z menu rozwijanego *Help* w środowisku IDLE, a także na stronie internetowej <http://www.python.org/doc>. Więcej informacji na temat dostępnych źródeł dokumentacji (włącznie z PyDoc i funkcją *help*) znajduje się w rozdziale 15. Jeżeli masz jeszcze chwilę wolnego czasu, możesz zatrzymać się na oficjalnej stronie Pythona, a także przejrzeć witrynę z repozytorium rozszerzeń zewnętrznych PyPy. Szczególną uwagę powinieneś zwrócić na dokumentację dostępną na stronie Python.org (<http://www.python.org>), gdzie znajdują się kluczowe zasoby dla każdego użytkownika zainteresowanego programowaniem w tym języku.

[1] Jak wspominaliśmy wcześniej przy okazji omawiania wierszy poleceń, wszystkie nowe wersje systemu Windows pozwalają na wpisywanie w wierszu polecenia samej nazwy pliku *.py* i wykorzystują rejestr systemowy do ustalenia, że plik powinienny zostać otwarty za pomocą Pythona (na przykład, wpisanie nazwy *brian.py* jest odpowiednikiem wpisania polecenia *python brian.py*). Ten tryb wiersza poleceń jest podobny do zapisu ze znakami *#!* z Uniksa, choć w systemie Windows dzieje się to dla całego systemu, a nie tylko dla pliku. Wymaga to również wyraźnego skojarzenia rozszerzenia *.py* z nazwą programu, który powinien taki plik uruchomić, w przeciwnym razie takie rozwiązywanie nie zadziała. Warto zauważyć, że niektóre programy systemu Windows mogą interpretować i wykorzystywać wiersz *shebang* ze znakami *#!* w podobny sposób jak w Uniksie, ale powłoka systemowa DOS po prostu go zignoruje.

[2] W razie potrzeby można również całkowicie zatrzymać wyskakiwanie okien konsoli DOS dla plików klikanych w systemie Windows, gdy nie chcesz wyświetlać wyników działania na ekranie. Pliki, których nazwy kończą się rozszerzeniem *.pyw*, wyświetlają jedynie okna tworzone przez skrypt, a nie domyślne okno wiersza poleceń konsoli. Pliki *.pyw* to po prostu pliki źródłowe *.py*, które w systemie Windows zachowują się w taki specjalny sposób. Są one wykorzystywane przede wszystkim w interfejsach użytkownika pisanych w Pythonie, które tworzą własne okna, często w połączeniu z różnymi technikami zapisywania wyników działania oraz błędów do plików. Jak wspominaliśmy już wcześniej, takie zachowanie Pythona jest możliwe po uprzednim skojarzeniu pliku *pythonw.exe* w wersji 3.2 i starszych (lub *pyw.exe* w wersji 3.3 i nowszych) do otwierania plików *.pyw*.

[3] Warto zauważyć, że *import* oraz *from* łączą się z nazwą pliku modułu (*myfile*) bez rozszerzenia *.py*. Jak pokażemy w części piątej książki, kiedy Python szuka pliku, wie, że w swoim procesie wyszukiwania musi uwzględnić rozszerzenie. Warto powtórzyć raz jeszcze: rozszerzenie musisz podawać w wierszu polecień powłoki systemu, jednak w instrukcjach *import* nie jest to konieczne.

[4] Informacja dla osób umierających z ciekawości: Python szuka zainportowanych modułów we wszystkich katalogach wymienionych w *sys.path* — liście łańcuchów znaków z nazwami katalogów znajdujących się w module *sys* Pythona, inicjalizowanej ze zmiennej środowiskowej *PYTHONPATH* — a także w zbiorze standardowych katalogów. Jeżeli chcemy zainportować moduł z katalogu innego od katalogu roboczego, musi on być wymieniony w ustawieniach *PYTHONPATH*. Więcej informacji na ten temat znajduje się w rozdziale 22.

[5] IDLE oficjalnie jest pochodną od środowiska IDE, ale tak naprawdę swą nazwę zawdzięcza członkowi grupy Monty Python — Ericowi Idle.

[6] Więcej informacji na temat osadzania Pythona w kodzie napisanym w językach C i C++ można znaleźć w książce *Programming Python* wydawnictwa O'Reilly. Interfejs API umożliwiający osadzenie kodu może bezpośrednio wywoływać funkcje Pythona, ładować jego moduły i wykonywać wiele innych operacji. Warto również zauważyć, że dystrybucja Jython pozwala na wywoływanie kodu Pythona przez programy napisane w Javie za pomocą API opartego na Javie (klasa interpretera języka Python).

Część II Typy i operacje

Rozdział 4. Wprowadzenie do typów obiektów Pythona

Niniejszy rozdział rozpoczyna naszą przygodę z językiem Python. W pewnym sensie w Pythonie „robimy coś z rzeczami”. To „coś” przybiera formę operacji (działan), takich jak dodawanie czy konkatenacja, natomiast „różne rzeczy” to obiekty, na których wykonujemy takie operacje. W tej części książki skupimy się właśnie na owych „różnych rzeczach”, jak również na tym, co mogą z nimi robić nasze programy.

Mówiąc bardziej formalnym językiem, w Pythonie dane przybierają postać *obiektów* — albo wbudowanych obiektów udostępnianych przez Pythona, albo obiektów tworzonych za pomocą klas Pythona lub innych narzędzi zewnętrznych, takich jak biblioteki rozszerzeń języka C. Choć nieco później znacznie tę definicję rozbudujemy, możemy przyjąć, że obiekty są generalnie obszarami pamięci zawierającymi określone wartości i posiadającymi zbiory powiązanych operacji. Jak się niebabem przekonasz, w skryptach Pythona *wszystko* jest obiektem; dotyczy to nawet zwykłych liczb (np. 99) i obsługiwanych operacji (dodawanie, odejmowanie itd.).

Ponieważ obiekty są najbardziej podstawowym elementem Pythona, rozpoczęniemy ten rozdział od przeglądu typów wbudowanych obiektów tego języka. W kolejnych rozdziałach ponownie będziemy powracać do tych zagadnień i bardziej szczegółowo omawiać tematy krótko poruszone w tym rozdziale. Tutaj naszym celem jest tylko krótkie przedstawienie podstawowych zagadnień.

Hierarchia pojęć w Pythonie

Zanim przejdziemy do kodu, warto jednak najpierw powiedzieć, jak niniejszy rozdział wpisuje się w ogólny obraz Pythona. Programy napisane w tym języku można rozłożyć na moduły, instrukcje, wyrażenia i obiekty w następujący sposób:

1. Programy składają się z modułów.
2. Moduły zawierają instrukcje i polecenia.
3. Instrukcje i polecenia zawierają wyrażenia.
4. Wyrażenia tworzą i przetwarzają obiekty.

Omówienie modułów zamieszczone w rozdziale 3. uwzględnia najwyższy poziom w tej hierarchii. Rozdziały tej części książki odwołują się do najniższego jej poziomu, czyli wbudowanych obiektów oraz wyrażeń, które tworzą i przetwarzają obiekty.

Instrukcje i polecenia będąmy omawiać w następnej części książki, choć przekonasz się, że są one w dużej mierze używane do zarządzania obiektami, które spotkamy tutaj. Co więcej, zanim dotrzemy do klas w tej bardziej zorientowanej obiektowo części książki, odkryjemy, że pozwalają one na definiowanie nowych, własnych typów obiektów, zarówno poprzez korzystanie z, jak i emulowanie obiektów, które będziemy tutaj omawiać. Z tego właśnie wzgledu

zagadnienia związane z wbudowanymi obiektami Pythona są obowiązkowym punktem podróży dla każdego użytkownika pragnącego programować w tym języku.

	<p>Tradycyjnie przyjęło się, że we wprowadzeniach do programowania często podkreślane są trzy filary: <i>sekwencje</i> („zrób najpierw to, potem tamto”), <i>wybory</i> („zrób to, jeżeli tamto jest prawdziwe”) i <i>powtarzanie</i> („zrób to wiele razy”). Python posiada narzędzia w każdej z tych trzech kategorii i dodaje jeszcze do tego narzędzia z czwartej kategorii, <i>definiowanie</i>, pozwalające na tworzenie funkcji i klas. Taki podział może Ci pomóc na wczesnym etapie poznawania języka, ale jest on jednak trochę sztuczny i nieco nadmiernie uproszczony. Na przykład niektóre wyrażenia stosowane w Pythonie możemy zakwalifikować zarówno jako powtórzenia, jak i wybory; niektóre z tych kategorii mają zupełnie inne znaczenia w Pythonie, a wiele pojęć, które poznasz nieco później, w ogóle nie pasuje do takiej formy podziału. W języku Python znacznie mocniej jednoczącym pojęciem są <i>obiekty</i> i to, co możemy z nimi zrobić. Już niebawem dowiesz się dlaczego.</p>
---	--

Dlaczego korzystamy z typów wbudowanych

Jeżeli używałeś już kiedyś języków niższego poziomu, takich jak C czy C++, doskonale zdajesz sobie sprawę z tego, że większość pracy programisty polega na implementowaniu *obiektów* — znanych również jako *struktury danych* (ang. *data structures*) — tak by reprezentowały odpowiednie komponenty składowe tworzonej aplikacji. Programista musi rozplanować struktury pamięci, zająć się procesem przydzielana pamięci czy zaimplementować procedury wyszukiwania i dostępu. Takie zadania są tak zmudne (i podatne na błędy), na jakie wyglądają, i zazwyczaj odwracają uwagę programisty od prawdziwych celów programu.

W typowych programach napisanych w Pythonie wykonywanie większości tych niewdzięcznych zadań jest całkowicie zbędne. Ponieważ Python udostępnia wiele typów obiektów jako nieodłączną część samego języka, zazwyczaj nie ma potrzeby dodatkowego implementowania obiektów przed rozpoczęciem rozwiązywania prawdziwych problemów. W rzeczywistości, o ile nie masz potrzeby korzystania ze specjalnych metod przetwarzania danych, które nie są dostępne w obiektach wbudowanych, prawie zawsze lepiej będzie skorzystać z wbudowanych obiektów, zamiast implementować własne. Poniżej znajduje się kilka przyczyn takiego stanu rzeczy.

- **Obiekty wbudowane sprawiają, że programy łatwo się pisze.** W przypadku prostych zadań obiekty wbudowane są często wszystkim, czego potrzebujesz do utworzenia struktur niezbędnych do działania programu. Od ręki otrzymujesz narzędzia o ogromnych możliwościach, takie jak kolekcje (listy) czy tabele, które można przeszukiwać (słowniki). Bardzo wiele zadań można wykonać, korzystając wyłącznie z samych obiektów wbudowanych.
- **Obiekty wbudowane są komponentami rozszerzeń.** W przypadku bardziej zaawansowanych zadań być może nadal konieczne będzie udostępnianie własnych obiektów, wykorzystywanie klas Pythona czy interfejsów języka C. Jednak jak okaże się w dalszej części książki, obiekty implementowane ręcznie są często zbudowane na bazie typów wbudowanych, takich jak listy czy słowniki. Na przykład strukturę danych stosu można zaimplementować jako klasę zarządzającą wbudowaną listą lub dostosowującą tę listę do własnych potrzeb.
- **Obiekty wbudowane często są bardziej wydajne od własnych struktur danych.** Wbudowane obiekty Pythona wykorzystują zoptymalizowane algorytmy struktur danych, które zostały zaimplementowane w języku C w celu zwiększenia szybkości ich działania.

Choć możesz samodzielnie utworzyć podobne typy obiektów, zazwyczaj trudno będzie Ci osiągnąć ten sam poziom wydajności, jaki udostępniają obiekty wbudowane.

- **Obiekty wbudowane są standardową częścią języka.** W pewien sposób Python zapożycza zarówno od języków opierających się na obiektach wbudowanych (jak na przykład LISP), jak i języków, w których to programista udostępnia implementacje własnych narzędzi czy struktur danych (jak w C++). Choć w Pythonie można implementować własne, unikatowe typy obiektów, nie musisz tego robić, by zacząć programować w tym języku. Co więcej, ponieważ obiekty wbudowane są standardem, zawsze pozostają takie same; własnościowe rozwiązania stosowane w wielu frameworkach zazwyczaj mają tendencję do zmian wraz z upływem czasu.

Innymi słowy, obiekty wbudowane nie tylko ułatwiają programowanie, ale mają także większe możliwości i są bardziej wydajne od większości obiektów tworzonych od podstaw. Bez względu na to, czy zdecydujesz się na zaimplementowanie nowych typów obiektów, wbudowane obiekty nadal będą stanowiły podstawę każdego programu napisanego w Pythonie.

Najważniejsze typy danych w Pythonie

W tabeli 4.1 zaprezentowano przegląd wbudowanych typów obiektów Pythona wraz ze składnią wykorzystywaną do kodowania ich *literałów* — czyli wyrażeń generujących takie obiekty^[1]. Niektóre z typów powinny dla osób znających inne języki programowania wyglądać znajomo. Liczby i łańcuchy znaków reprezentują, odpowiednio, wartości liczbowe i tekstowe, a obiekty plikowe zapewniają interfejsy pozwalające na przetwarzanie plików.

Dla wielu użytkowników typy obiektów przedstawione w tabeli 4.1 są bardziej ogólne i mają większe możliwości, niż bywa to w innych językach. Na przykład przekonasz się, że już same listy i słowniki mają ogromne możliwości reprezentowania danych, które zwalniają programistę z konieczności wykonywania większości zadań niezbędnych zazwyczaj do zaimplementowania obsługi kolekcji i wyszukiwania w językach niższego poziomu. Mówiąc w skrócie, listy udostępniają uporządkowane zbiory innych obiektów, natomiast słowniki przechowują obiekty według określonego klucza. Zarówno listy, jak i słowniki mogą być zagnieżdżane, mogą rosnąć i kurczyć się na życzenie oraz mogą zawierać obiekty dowolnego typu.

Tabela 4.1. Przegląd wbudowanych obiektów Pythona

Typ obiektu	Przykładowe literały (tworzenie)
Liczby	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Łańcuchy znaków	'mielonka', "Brian", b'a\x01c', u'sp\xc4m'
Listy	[1, [2, 'trzy'], 4.5], list(range(10))
Słowniki	{'jedzenie': 'mielonka', 'smak': 'mniam'}, dict(godziny=10)
Krotki	(1, 'mielonka', 4, 'U'), tuple('mielonka'), namedtuple
Pliki	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Zbiory	set('abc'), {'a', 'b', 'c'}
Inne typy podstawowe	Wartości Boolean, typy, None

Typy jednostek programu	Funkcje, moduły, klasy (część IV, V i VI książki)
Typy powiązane z implementacją	Kod skompilowany, ślady stosu (część IV i VII książki)

Wymienione w tabeli 4.1 *jednostki programów*, takie jak funkcje, moduły i klasy, które będziemy spotykać w dalszej części książki, w języku Python są obiektami, tworzonymi za pomocą instrukcji oraz wyrażeń takich jak `def`, `class`, `import` czy `lambda`, i można je swobodnie przekazywać w skryptach bądź przechowywać w innych obiektach. Python udostępnia również zbiór typów *powiązanych z implementacją*, takich jak obiekty skompilowanego kodu, które są zazwyczaj bardziej przedmiotem zainteresowania programistów tworzących narzędzia niż twórców aplikacji; będą o nich wspominać również w późniejszych częściach książki, choć w mniejszym stopniu ze względu na ich wyspecjalizowane role.

Pomimo swojego nagłówka tabela 4.1 nie jest całkowicie kompletna, ponieważ *wszystko*, co przetwarzamy w programach Pythona, jest takimi czy innymi obiektami. Na przykład, kiedy w Pythonie wykonujemy dopasowywanie tekstu do wzorca, tworzymy obiekty wzorców, a gdy tworzymy skrypty sieciowe, używamy obiektów reprezentujących gniazda sieciowe. Te pozostałe typy obiektów tworzone są głównie poprzez importowanie i używanie funkcji z modułów bibliotecznych (na przykład modułu `re` dla wzorców tekstu czy `socket` dla gniazd sieciowych) i z reguły mają swoje własne, niestandardowe typy zachowania.

Pozostałe typy obiektów z tabeli 4.1 nazywane są zazwyczaj *typami podstawowymi*, ponieważ są one tak naprawdę wbudowane w język, co znaczy, że istnieje odpowiednia składnia wyrażeń do generowania większości z nich. Na przykład, kiedy uruchomimy następujący kod zawierający ciąg znaków ujęty w apostrofy...:

```
>>> 'mielonka'
```

...z technicznego punktu widzenia wykonujemy wyrażenie z literałem, które generuje i zwraca nowy obiekt typu `string`. W Pythonie istnieje specyficzna składnia wyrażenia, która tworzy ten obiekt. Podobnie, wyrażenie umieszczone w nawiasach kwadratowych tworzy *listę*, a w nawiasach klamrowych — *słownik*. Choć — jak się niedługo okaże — w Pythonie nie używamy deklarowania typów, składnia wykonywanego wyrażenia określa typy tworzonych i wykorzystywanych obiektów. Wyrażenia generujące obiekty, jak te z tabeli 4.1, to właśnie miejsca, z których pochodzą typy obiektów.

Co równie ważne: kiedy tworzymy jakiś obiekt, wiążemy go z określonym zbiorem operacji. Na łańcuchach znaków można wykonywać tylko operacje dostępne dla łańcuchów znaków, a na listach — tylko operacje przeznaczone dla list. Pod względem formalnym oznacza to, że Python jest językiem z *typami dynamicznymi* (co znaczy, że automatycznie przechowuje informacje o typach, zamiast wymagać kodu z deklaracją typu), ale jednocześnie jego typy są *silne* (to znaczy na obiekcie można wykonać tylko te operacje, które są poprawne dla określonego typu).

W kolejnych rozdziałach szczegółowo omówimy poszczególne typy obiektów zaprezentowane w tabeli 4.1. Zanim jednak zagłębimy się w szczegóły, najpierw przyjrzyjmy się podstawowym obiektom Pythona w działaniu. Pozostała część rozdziału zawiera przegląd operacji, które bardziej dokładnie omówimy w kolejnych rozdziałach. Nie należy oczekwać, że zostanie tutaj zaprezentowane wszystko — celem niniejszego rozdziału jest tylko zaostrzenie Twojego apetytu i wprowadzenie pewnych kluczowych koncepcji. Najlepszym sposobem na rozpoczęcie czegoś jest... samo rozpoczęcie, zatem czas zabrać się za prawdziwy kod.

Liczby

Dla osób, które zajmowały się już programowaniem czy tworzeniem skryptów, niektóre typy danych z tabeli 4.1 będą wyglądały znajomo. Nawet dla osób niemających nic wspólnego z programowaniem liczby wyglądają dość prosto. Zbiór podstawowych obiektów Pythona obejmuje typowe rodzaje liczb: *całkowite* (liczby bez części ułamkowej), *zmiennoprzecinkowe* (liczby z częścią po przecinku), a także bardziej egzotyczne typy liczbowe (*liczby zespolone*, *liczby stałoprzecinkowe*, *liczby wymierne* z mianownikiem i licznikiem, a także pełne zbiory). Wbudowane rodzaje liczb są w zupełności wystarczające, aby reprezentować większość wartości liczbowych — od Twojego wieku, po saldo w banku, niemniej warto pamiętać, że istnieje jeszcze więcej typów liczb, implementowanych jako dodatki tworzone przez niezależnych programistów.

Choć oferują kilka bardziej zaawansowanych opcji, podstawowe typy liczbowe Pythona są... właściwie podstawowe. Liczby w Pythonie obsługują normalne działania matematyczne. Na przykład znak + wykonuje dodawanie, znak * mnożenie, natomiast ** potęgowanie.

```
>>> 123 + 222                                # Dodawanie liczb całkowitych
345
>>> 1.5 * 4                                    # Mnożenie liczb
zmiennoprzecinkowych
6.0
>>> 2 ** 100                                   # 2 do potęgi 100
1267650600228229401496703205376
```

Warto zwrócić uwagę na wynik ostatniego działania. Typ reprezentujący liczby całkowite w Pythonie 3.x automatycznie udostępnia dodatkową precyzję dla tak dużych liczb, kiedy jest to potrzebne (w Pythonie 2.x liczby zbyt duże dla zwykłej liczby całkowitej typu *integer* były obsługiwane przez osobny typ *long integer*). Można na przykład obliczyć w Pythonie 2 do potęgi 1 000 000 jako liczbę całkowitą (choć pewnie raczej nie powinieneś próbować wyświetlać wyniku takiej operacji działania — wyświetlanie liczby mającej ponad trzysta tysięcy cyfr z pewnością zajmie trochę czasu!).

```
>>> len(str(2 ** 1000000))                  # Ile cyfr będzie w naprawdę
DUŻEJ liczbie?
301030
```

Taka forma zagnieżdżonego wywołania funkcji działa w kierunku od wewnątrz na zewnątrz — najpierw wynik potęgowania ** przekształcany jest na ciąg cyfr za pomocą wbudowanej funkcji *str*, a następnie za pomocą funkcji *len* obliczana jest długość wynikowego ciągu znaków. Końcowym rezultatem takiego polecenia jest wartość reprezentująca liczbę cyfr w liczbie będącej wynikiem potęgowania. Funkcje *str* i *len* działają na wielu typach obiektów; więcej szczegółowych informacji na temat tych funkcji znajdziesz w dalszej części książki.

Kiedy w Pythonie w wersjach wcześniejszych niż 2.7 i 3.1 zaczniesz eksperymentować z liczbami zmiennoprzecinkowymi, prawdopodobnie natkniesz się na coś, co na pierwszy rzut oka może wyglądać trochę dziwnie:

```
>>> 3.1415 * 2                                # repr: pełna postać (Python <2.7
i 3.1)
6.2830000000000004
>>> print(3.1415 * 2)                          # str: postać przyjazna dla
użytkownika
6.283
```

Pierwszy wynik nie jest błędem — to kwestia sposobu wyświetlania. Okazuje się, że każdy obiekt można wyświetlić na dwa sposoby — z pełną precyją (jak w pierwszym wyniku powyżej) oraz w formie przyjaznej dla użytkownika (jak w drugim wyniku). Formalnie rzecz biorąc, pierwsza postać znana jest jako postać drukowalna obiektu (`repr`), natomiast druga jest postacią przyjazną dla użytkownika (`str`). W starszych Pythonach wartości zmiennoprzecinkowe były czasami w pełnej postaci wyświetlane z większą precyją, niż mógłbyś tego oczekwać. Różnica ta zacznie mieć znaczenie, kiedy przejdziemy do używania klas. Na razie, kiedy coś będzie dziwnie wyglądało, spróbuj wyświetlić to za pomocą wywołania wbudowanej funkcji `print`.

Jeszcze lepszym rozwiązaniem będzie aktualizacja Pythona do wersji 2.7 lub najnowszej serii 3.x, gdzie liczby zmiennoprzecinkowe wyświetlane są w bardziej intelligentny sposób, zwykle z mniejszą liczbą cyfr po przecinku. Ponieważ ta książka jest oparta na Pythonie w wersji 2.7 i 3.3, to liczby zmiennoprzecinkowe będziemy prezentować w następującej postaci:

```
>>> 3.1415 * 2                                # repr: pełna postać (Python >=
2.7 i 3.1)
6.283
```

Oprócz wyrażeń w Pythonie dostępnych jest kilka przydatnych modułów do obliczeń numerycznych. *Moduły* są po prostu pakietami dodatkowych narzędzi, które musimy zaimportować, aby móc z nich skorzystać.

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.sqrt(85)
9.2195444572928871
```

Moduł `math` zawiera bardziej zaawansowane narzędzia numeryczne, takie jak funkcje, natomiast moduł `random` pozwala na generowanie liczb losowych, a także losowe wybieranie elementu z listy (w naszym przypadku jest to *lista* elementów umieszczonych w nawiasach kwadratowych, czyli uporządkowana kolekcja obiektów; będąmy o nich mówić w dalszej części rozdziału):

```
>>> import random
>>> random.random()
0.7082048489415967
>>> random.choice([1, 2, 3, 4])
1
```

Python udostępnia również bardziej egzotyczne obiekty liczbowe, takie jak liczby zespolone, liczby stałoprzecinkowe, liczby wymierne, a także zbiory i wartości typu Boolean (wartości logiczne). Można również znaleźć różne dodatkowe rozszerzenia na licencji open source (na przykład macierze, wektory czy liczby o rozszerzonej precyji). Szczegółowe omówienie takich typów zostawimy sobie na później.

Jak na razie używaliśmy Pythona w postaci prostego kalkulatora; aby jednak oddać sprawiedliwość jego typom wbudowanym, przejdziemy teraz do omówienia łańcuchów znaków.

Łańcuchy znaków

Łańcuchy znaków (ang. *strings*) są wykorzystywane zarówno do przechowywania informacji tekstowych, jak i dowolnych kolekcji bajtów (takich jak na przykład zawartość pliku graficznego). Są pierwszym przykładem tego, co w Pythonie znane jest pod nazwą *sekwencji* — czyli uporządkowanych kolekcji innych obiektów. Sekwencje zachowują porządek przechowywanych elementów od lewej do prawej strony. Elementy sekwencji są przechowywane i pobierane według ich pozycji względnych. Ścisłe mówiąc, łańcuchy znaków są sekwencjami ciągów jednoznakowych; inne, bardziej ogólne typy sekwencji obejmują *listy* i *krotki*, które zostaną opisane nieco później.

Operacje na sekwencjach

Będąc sekwencjami, łańcuchy znaków obsługują operacje zakładające pozycyjne uporządkowanie elementów. Na przykład, jeżeli mamy łańcuch składający się z ośmiu znaków (zazwyczaj w postaci ciągu znaków ujętego w apostrofy lub cudzysłów), możemy zweryfikować jego długość za pomocą wbudowanej funkcji `len` i pobrać jego elementy za pomocą wyrażeń *indeksujących*:

```
>>> S = 'Mielonka'                                # Tworzy czteroznakowy ciąg znaków i
przypisuje go do zmiennej

>>> len(S)                                         # Długość

8

>>> S[0]                                           # Pierwszy element w S, indeksowanie
rozpoczyna się od zera

'M'

>>> S[1]                                           # Drugi element od lewej

'i'
```

W Pythonie indeksy są kodowane jako wartość przesunięcia liczona od początku łańcucha, dlatego rozpoczynają się od zera. Pierwszy element ma indeks 0, kolejny element ma indeks 1 i tak dalej.

Warto tutaj zwrócić uwagę na przypisanie łańcucha znaków do *zmiennej* o nazwie `S`. Szczegółowe omówienie tego, jak to działa, odłożymy na później (a w szczególności do rozdziału 6.), natomiast teraz powinieneś wiedzieć, że zmiennych Pythona nigdy nie trzeba deklarować z wyprzedzeniem. Zmienna tworzona jest w momencie, kiedy przypisujemy do niej wartość; można do niej przypisać dowolny typ obiektu i zostanie zastąpiona swoją wartością, kiedy pojawi się w wyrażeniu. Zanim będziesz mógł użyć zmiennej, musisz do niej przypisać jakąś wartość. Na potrzeby niniejszego rozdziału wystarczy zapamiętać, że jeżeli chcesz przechować dany obiekt do późniejszego użycia, musisz taki obiekt przypisać do zmiennej.

W Pythonie można również indeksować od końca — indeksy dodatnie liczone są od lewej strony do prawej, natomiast ujemne od prawej do lewej:

```
>>> S[-1]                                         # Pierwszy element od końca zmiennej S

'a'

>>> S[-2]                                         # Drugi element od końca zmiennej S

'k'
```

Z formalnego punktu widzenia indeks ujemny jest po prostu dodawany do rozmiaru łańcucha znaków, dzięki czemu dwie poniższe operacje są równoważne (choć pierwszą łatwiej jest zapisać, a trudniej się w niej pomylić):

```

>>> S[-1]                                # Ostatni element w zmiennej S
'a'

>>> S[len(S)-1]                         # Indeksowanie ujemne, trudniejszy
sposób zapisu

'a'

```

Warto zauważyć, że w nawiasach kwadratowych możemy wykorzystać *dowolne wyrażenie*, a nie tylko zakodowany na stałe literał liczbowy. W każdym miejscu, w którym Python oczekuje wartości, można użyć literaka, zmiennej lub dowolnego wyrażenia. Składnia Pythona jest w tym zakresie bardzo ogólna.

Poza prostym indeksowaniem zgodnie z pozycją, sekwencje obsługują również bardziej ogólną formę indeksowania znaną jako *wycinki* (ang. *slice*). Polega ona na ekstrakcji całego podcięgu znaków (wycinka) za jednym razem, na przykład:

```

>>> S                                     # Łańcuch z ośmioma znakami
'Mielonka'

>>> S[1:3]                               # Wycinek z S z indeksami od 1 do 2
(bez 3)

'ie'

```

Wycinki najłatwiej jest sobie wyobrazić jako sposób pozwalający na ekstrakcję całej *kolumny* z łańcucha znaków za jednym razem. Ich ogólna forma, $X[I:J]$, oznacza: „zwróć wszystkie znaki z ciągu X od przesunięcia I aż do przesunięcia J, ale bez tego ostatniego elementu”. Wynik zwracany jest w postaci nowego obiektu. Na przykład druga operacja z powyższego przykładu zwraca wszystkie znaki z ciągu S od przesunięcia 1 do przesunięcia 2 (czyli 3-1) jako nowy łańcuch znaków. Wynikiem jest wycinek składający się z dwóch znaków znajdujących się w środku łańcucha S.

W wycinkach lewą granicą jest domyślnie zero, natomiast prawą — długość sekwencji, z której coś wycinamy. Dzięki temu można spotkać różne warianty użycia wycinków:

```

>>> S[1:]                                 # Wszystko poza pierwszym znakiem
(1:len(S))

'ielonka'

>>> S                                     # Łańcuch S się nie zmienił
'Mielonka'

>>> S[0:7]                               # Wszystkie elementy bez ostatniego
'Mielonk'

>>> S[:7]                                 # To samo co S[0:7]
'Mielonk'

>>> S[::-1]                             # Wszystkie elementy bez ostatniego,
ale w łatwiejszej postaci(0:-1)
'Mielonk'

>>> S[:]                                 # Całość S jako kopia najwyższego
poziomu (0:len(S))

'Mielonka'

```

Zwróć uwagę na to, że do wyznaczenia granic wycinków mogą również zostać wykorzystane ujemne wartości przesunięcia (jak to zostało zrobione w przedostatnim wyrażeniu), a wynikiem ostatniej operacji jest skopiowanie całego łańcucha znaków. Jak się później przekonasz, takie kopiowanie łańcucha znaków nie ma większego sensu, choć taka forma polecenia może się przydać w przypadku sekwencji takich jak listy.

Będąc sekwencjami, łańcuchy znaków obsługują również *konkatenację* (ang. *concatenation*) za pomocą znaku + (czyli łączenie dwóch łańcuchów znaków w jeden ciąg), a także *powtórzenia* (ang. *repetition*), czyli budowanie nowego łańcucha znaków poprzez powtórzenie innego:

```
>>> S
'Mielonka'
>>> S + 'xyz'                                # Konkatenacja
'Mielonkaxyz'
>>> S                                         # S pozostaje bez zmian
'Mielonka'
>>> S * 8                                     # Powtórzenie
'MielonkaMielonkaMielonkaMielonkaMielonkaMielonkaMielonka'
```

Zwróć uwagę na to, że znak plus (+) dla różnych obiektów może oznaczać różne operacje — dodawanie dla liczb, a konkatenację dla łańcuchów znaków. Jest to właściwość Pythona, którą w dalszej części książki będziemy nazywali *polimorfizmem*. Oznacza to, że działanie danej operacji uzależnione jest od typu obiektów, na jakich się ją wykonuje. Jak się przekonasz przy okazji omawiania typowania dynamicznego, to właśnie polimorfizm odpowiada za zwięzość i elastyczność kodu napisanego w Pythonie. Ponieważ nie mamy ograniczeń typów, operacja napisana w Pythonie może zazwyczaj automatycznie działać na wielu różnych typach obiektów, o ile obsługują one zgodny z nią interfejs (jak operacja + powyżej). W Pythonie jest to bardzo ważna koncepcja, do której jeszcze wrócimy.

Niezmiennosć

Warto zauważyć, że w poprzednich przykładach żadna operacja wykonana na łańcuchu znaków nie zmieniła oryginalnego łańcucha. Każda operacja na łańcuchach znaków zdefiniowana jest w taki sposób, by w rezultacie zwracać nowy łańcuch znaków, ponieważ łańcuchy są w Pythonie *niezmienne* (ang. *immutable*). Nie mogą być zmieniane w miejscu już po utworzeniu, czyli na przykład nie można zmienić łańcucha znaków, przypisując go do jednej z jego pozycji; inaczej mówiąc, nie można nadpisać wartości obiektów niezmiennych, ale zawsze można stworzyć nowy łańcuch znaków i przypisać go do tej samej nazwy (zmiennej). Ponieważ Python czyści stare obiekty w miarę przechodzenia dalej (o czym przekonamy się nieco później), nie jest to aż tak niewydajne, jak mogłoby się wydawać.

```
>>> S
'Mielonka'
>>> S[0] = 'z'                                # Niezmienne obiekty nie mogą
być modyfikowane
...pominieto treść komunikatu o błędzie...
TypeError: 'str' object does not support item assignment
>>> S = 'z' + S[1:]                           # Można jednak tworzyć wyrażenia
budujące nowe obiekty
```

```
>>> S  
'zielonka'
```

Każdy obiekt Pythona klasyfikowany jest jako niezmienny (niemodyfikowalny) bądź zmienny (modyfikowalny). Wśród typów podstawowych niezmienne są *liczby*, *łańcuchy znaków* oraz *krotki*. *Listy*, *słowniki* i *zbiory* można dowolnie zmieniać w miejscu, podobnie jak większość obiektów, które tworzysz za pomocą klas. Takie rozróżnienie okazuje się być kluczowe w sposobie działania Pythona, ale na szczególną odpowiedź, dlaczego tak się dzieje, będziesz musiał jeszcze chwilę poczekać. Niezmienność można między innymi wykorzystać do zagwarantowania, że obiekt pozostanie stały w czasie całego cyklu działania programu; modyfikowalne (niestałe) wartości obiektów mogą się zmieniać w dowolnym czasie i miejscu (niezależnie, czy tego oczekujesz, czy nie).

Mówiącścielj, dane tekstowe można modyfikować w *miejscu*, jeżeli zamienisz je na listę pojedynczych znaków i połączysz z powrotem bez używania separatorów lub użyjesz nowszego typu *bytearray*, dostępnego w Pythonach 2.6, 3.0 i nowszych:

```
>>> S = 'taczka'  
>>> L = list(S) # Zamień na listę: [...]  
>>> L  
['t', 'a', 'c', 'z', 'k', 'a']  
>>> L[1] = 'p' # Modyfikuj w miejscu  
>>> ''.join(L) # Połącz bez użycia znaku  
separatorka  
'paczka'  
>>> B = bytearray(b'mini') # Hybryda zbioru bajtów i listy  
>>> B.extend(b'maraton') # 'b' jest potrzebne w wersji  
3.x, ale w 2.x już nie  
>>> B # B[i] = ord(x) też tutaj będzie  
działać  
bytearray(b'minimaraton')  
>>> B.decode() # Zamień na normalny ciąg znaków  
'minimaraton'
```

Typ *bytearray* obsługuje zmiany w miejscu dla tekstu, ale tylko dla znaków mających maksymalnie 8 bitów (np. ASCII). Wszystkie inne ciągi są nadal niezmienne — *bytearray* jest wyraźną hybrydą niezmiennych ciągów bajtów (których składnia `b'...'` jest wymagana w wersji 3.x i opcjonalna w wersji 2.x) oraz modyfikowalnych list (definiowanych i wyświetlanych w nawiasach kwadratowych `[]`). Aby móc w pełni zrozumieć, jak to działa, musimy dowiedzieć się jak kodowane są znaki ASCII i Unicode.

Metody specyficzne dla typu

Każda z omówionych dotychczas operacji na łańcuchach znaków jest tak naprawdę operacją na sekwencjach — takie operacje będą działały również na innych typach sekwencji w Pythonie, w tym na listach i krotkach. Poza operacjami wspólnymi dla sekwencji, łańcuchy znaków obsługują również własne operacje dostępne w formie *metod* (czyli funkcji dołączanych do określonego obiektu i wywoływanych za pomocą odpowiedniego wyrażenia).

Na przykład metoda `find` jest podstawową operacją wyszukiwania podciągów znaków działającą na łańcuchach znaków (zwraca wartość przesunięcia znalezionej podcięgu znaków lub `-1`, jeżeli taki podciąg nie jest obecny). Z kolei metoda `replace` służy do globalnego wyszukiwania i zastępowania ciągów znaków; obie metody działają na obiekcie, do którego są dołączone i z którego są wywoływane.

```
>>> S = 'Mielonka'  
>>> S.find('ie')                                     # Odnalezienie przesunięcia  
podłańcucha w S  
1  
>>> S  
'Mielonka'  
>>> S.replace('ie', 'XYZ')                         # Zastąpienie wystąpień  
podłańcucha innym  
'MXYZlonka'  
>>> S  
'Mielonka'
```

I znów, mimo że w metodach łańcucha znaków występuje nazwa zmiennej, nie zmieniamy oryginalnych łańcuchów znaków, lecz tworzymy nowe. Ponieważ łańcuchy znaków są niezmienne, jest to jedyny sposób, w jaki może to działać. Metody działające na łańcuchach znaków to podstawowe narzędzia do przetwarzania tekstu w Pythonie. Istnieją również inne metody pozwalające dzielić ciągi znaków na podłańcuchy w miejscu wystąpienia separatora (co przydaje się w analizie składniowej), zmieniać wielkość liter, sprawdzać zawartość łańcuchów znaków (cyfry, litery i inne znaki) czy usuwać białe znaki (ang. *whitespace*) z końca łańcucha.

```
>>> line = 'aaa,bbb,cccc,dd'  
>>> line.split(',')                                # Podział na podcięgi według  
znaków separatora  
['aaa', 'bbb', 'cccc', 'dd']  
>>> S = 'mielonka'  
>>> S.upper()                                      # Konwersja na wielkie litery  
'MIELONKA'  
>>> S.isalpha()                                    # Sprawdzenie zawartości:  
isalpha, isdigit itd.  
True  
>>> line = 'aaa,bbb,cccc,dd\n'  
>>> line = line.rstrip()                           # Usunięcie białych znaków po  
prawej stronie  
'aaa,bbb,cccc,dd'  
>>> line.rstrip().split(',')                      # Połączenie dwóch operacji  
['aaa', 'bbb', 'cccc', 'dd']
```

Zwróci uwagę na ostatnie polecenie w przykładzie powyżej, które najpierw usuwa spacje z prawej strony tekstu, a dopiero potem dzieli串 znaków na podcięgi według separatora. Dzieje

się tak, ponieważ wyrażenia Pythona wykonywane są od lewej do prawej. Łańcuchy znaków obsługują również zaawansowane operacje zastępowania znane jako *formatowanie*, dostępne zarówno w postaci wyrażeń (oryginalne rozwiązywanie), jak i wywołania metody łańcuchów znaków (nowość w wersjach 2.6 oraz 3.0); w wersjach 2.7, 3.1 i nowszych można w wywołaniu metody pominąć względne wartości liczbowe argumentów:

```
>>> '%s, jajka i %s' % ('mielonka', 'MIELONKA!')           # Wyrażenie
formatujące (wszystkie wersje)
'mielonka, jajka i MIELONKA!'

>>> '{0}, jajka i {1}'.format('mielonka', 'MIELONKA!') # Metoda formatująca
(wersje 2.6+ i 3.0+)
'mielonka, jajka i MIELONKA!'

>>> '{}, jajka i {}'.format('mielonka', 'MIELONKA!')   # Wartości liczbowe są
opcjonalne

# (wersje 2.7+, 3.1+)
'mielonka, jajka i MIELONKA!'
```

Mechanizm formatowania łańcuchów tekstu w Pythonie posiada ogromne możliwości, o których będziemy mówić (przynajmniej częściowo) w dalszych rozdziałach książki. Zazwyczaj formatowanie tekstu ma największe znaczenie podczas generowania raportów zawierających dane liczbowe.

```
>>> '{:,.2f}'.format(296999.2567)                      # Separator, miejsca po przecinku
'296,999.26'
>>> '%.2f | %+05d' % (3.14159, -42)                 # Cyfry, dopełnienie, znaki
'3.14 | -0042'
```

Jedna uwaga: operacje na sekwencjach są uniwersalne, ale metody takie nie są i choć niektóre typy obiektów posiadają metody o takich samych nazwach, to powinieneś pamiętać, że metody łańcuchów znaków będą działać wyłącznie na łańcuchach znaków i na niczym innym. Generalnie zbiór narzędzi Pythona posiada strukturę warstwową — operacje uniwersalne, które są dostępne dla większej liczby typów danych, występują jako wbudowane funkcje lub wyrażenia (na przykład `len(X)`, `X[0]`), natomiast operacje specyficzne dla określonego typu są wywołaniami metod (jak `aString.upper()`). Odnalezienie odpowiednich narzędzi w poszczególnych kategoriach stanie się z czasem coraz łatwiejsze i bardziej naturalne, a w kolejnym podrozdziale znajdziesz się kilka wskazówek przydatnych już teraz.

Uzyskiwanie pomocy

Metody wprowadzone w poprzednim podrozdziale są reprezentatywną, choć dość niewielką próbką tego, co dostępne jest dla łańcuchów znaków. Generalnie niniejsza książka nie zawiera kompletnego omówienia metod obiektów. Aby uzyskać więcej szczegółowych informacji, zawsze możesz wywołać wbudowaną funkcję `dir`, która wywołana bez argumentów zwraca listę nazw z lokalnej tablicy symboli, a wywołana z argumentem (co jest znacznie bardziej użyteczne) wyświetla listę wszystkich atrybutów wskazanego obiektu. Ponieważ metody są atrybutami obiektów, zostaną wyświetlone na tej liście. Zakładając, że zmienna `S` nadal jest łańcuchem znaków, lista jej atrybutów w Pythonie 3.3 wygląda następująco (w Pythonie 2.x lista atrybutów będzie się nieco różnić):

```
>>> dir(S)
```

```
[ '__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__formatter_field_name_split__',
 '__formatter_parser__', 'capitalize', 'casfold', 'center', 'count', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Nazwami z *podwójnymi znakami podkreślenia* (`__`) nie musisz się teraz przejmować; powrócimy do nich w dalszej części książki, kiedy będziemy omawiać przeciążanie operatorów w klasach — takie nazwy reprezentują implementację obiektu łańcucha znaków i pozwalają na dostosowywanie obiektów do własnych potrzeb. Na przykład metoda `__add__` ciągu znaków jest tym, co naprawdę wykonuje konkatenację; w rzeczywistości Python wewnętrznie zamienia pierwsze wyrażenie pokazane poniżej do postaci drugiego, chociaż zwykle nie powinieneś samodzielnie używać tej drugiej formy (jest mniej intuicyjna, a w dodatku może nawet działać wolniej):

```
>>> S + 'NI!'
'spamNI!'
>>> S.__add__('NI!')
'spamNI!'
```

Ogólna reguła jest taka, że podwójne znaki podkreślenia (`__`) na początku i końcu nazwy to konwencja wykorzystywana w Pythonie do szczegółów implementacyjnych. Nazwy bez tych znaków są metodami, które można wywoływać na obiektach reprezentujących łańcuchy znaków.

Funkcja `dir` wyświetla same nazwy metod. Aby dowiedzieć się, co te metody robią, możesz przekazywać ich nazwy do funkcji `help`.

```
>>> help(S.replace)
Help on built-in function replace:

replace(...)

S.replace (old, new[, count]) -> str
Return a copy of string S with all occurrences of substring
old replaced by new. If the optional argument count is
given, only the first count occurrences are replaced.
```

Funkcja `help` jest jednym z kilku interfejsów do pakietu udostępnianego wraz z Pythonem, znanego pod nazwą *PyDoc*, czyli narzędzia służącego do pobierania dokumentacji z obiektów. W dalszej części książki pokażemy, że PyDoc może również zwracać swoje raporty w formacie HTML, dzięki czemu możesz wyświetlać je w przeglądarce sieciowej.

Można również próbować uzyskać pomoc dla *całego łańcucha znaków* (na przykład `help(S)`), jednak w rezultacie nie zawsze możesz otrzymać to, co chciałbyś zobaczyć — w starszych wersjach Pythona wyświetcone zostaną informacje o wszystkich metodach dla łańcuchów znaków, a w nowszych wersjach prawdopodobnie nie otrzymasz żadnej pomocy, ponieważ ciągi

znaków są w nich traktowane w specjalny sposób. Zazwyczaj znacznie lepszym rozwiązaniem jest zapytanie o określona metodę, jak pokazano na listingu powyżej.

Zarówno funkcja `dir`, jak i `help` akceptują jako argumenty albo prawdziwy obiekt (jak nasz ciąg `S`), albo nazwę typu danych (jak `str`, `list` i `dict`). Ta ostatnia forma zwraca tę samą listę, co funkcja `dir`, ale uzupełnioną o pełne informacje o typie dla funkcji `help` i pozwala zapytać o konkretną metodę poprzez nazwę typu (np. pomoc dla metody `str.replace`).

Więcej szczegółowych informacji można również znaleźć w dokumentacji biblioteki standardowej Pythona lub wielu książkach, jednak to właśnie funkcje `dir` oraz `help` są w Pythonie pierwszym źródłem pomocy.

Inne sposoby kodowania łańcuchów znaków

Dotychczas przyglądaliśmy się operacjom na obiektach reprezentujących łańcuchy znaków, a także metodom przeznaczonym dla tego typu danych. Python udostępnia również różne sposoby kodowania łańcuchów znaków, które bardziej szczegółowo omówione zostaną nieco później. Na przykład znaki specjalne mogą być wpisywane jako sekwencje z lewym ukośnikiem spełniającym rolę znaku ucieczki; takie sekwencje są w Pythonie prezentowane w formacie szesnastkowym `\xNN`, chyba że reprezentują znaki drukowalne:

```
>>> S = 'A\nB\tC'                      # \n oznacza znak końca wiersza, a \t to tabulator
>>> len(S)                            # Każdy z elementów składowych S to jeden znak
5
>>> ord('\n')                          # \n to jeden znak zakodowany jako wartość dziesiętna 10
10
>>> S = 'A\0B\0C'                      # \0, binarny bajt zerowy, nie kończy łańcucha znaków
>>> len(S)                            # 5
>>> S                                # Znaki niedrukowalne wyświetlane są jako sekwencje szesnastkowe \xNN
'A\x00B\x00C'
```

Python pozwala na umieszczanie łańcuchów znaków zarówno w *cudzysłowach*, jak i *apostrofach* — oznaczają one to samo, ale pozwalają na osadzanie innego rodzaju cudzysłowu bez konieczności stosowania znaku ucieczki (większość programistów woli używać apostrofów). Python pozwala również na używanie wielowierszowych literałów łańcuchowych, umieszczanych w trzech cudzysłowach lub trzech apostrofach — gdy taka forma zostanie użyta, wszystkie wiersze składowe są ze sobą łączone, a w miejscach podziału wierszy dodawane są odpowiednie znaki końca wiersza. Jest to pewna konwencja syntaktyczna, która przydaje się do osadzania wielowierszowych fragmentów kodu HTML, XML czy JSON w skrypcie Pythona lub tymczasowego „wyłączania” wybranych fragmentów kodu — aby to zrobić, wystarczy dodać po trzy znaki cudzysłowu powyżej i poniżej takiego fragmentu:

```
>>> msg = """aaaaaaaaaaaaaa
bbb'''bbbbbbbbbb'''bbbbbbbb 'bbbb
```

```

cccccccccccccc
.....
>>> msg
'\naaaaaaaaaaaa\nbbb\''\bBBBBBBBBB"bbbbbbb\bbbb\ncccccccccccc\'

```

Python obsługuje również „surowe” literały łańcuchowe, które wyłączają mechanizm znaków ucieczki z lewym ukośnikiem. Takie literały rozpoczynają się od litery `r` i są bardzo przydatne do zapisywania ciągów takich jak ścieżki do plików w systemie Windows (na przykład `r'C:\tekst\nowy'`).

Ciągi znaków w formacie Unicode

Python wyposażony jest również w pełną obsługę ciągów znaków w formacie *Unicode*, który jest wymagany do przetwarzania tekstu w międzynarodowych zestawach znaków. Na przykład znaki w alfabetie japońskim i rosyjskim znajdują się poza standardowym zestawem znaków ASCII. Znaki inne niż ASCII mogą pojawiać się na stronach internetowych, w wiadomościach e-mail, graficznych interfejsach użytkownika, plikach JSON, XML i w wielu innych miejscach. Kiedy tak się dzieje, poprawne działanie programu wymaga obsługi znaków w formacie Unicode. Python z zasady ma taką wbudowaną obsługę, ale działa ona nieco inaczej w różnych jego wersjach; jest to jedna z najważniejszych różnic pomiędzy liniami 2.x i 3.x Pythona.

W *Pythonie 3.x* normalny ciąg znaków `str` obsługuje tekst Unicode (w tym ASCII, który jest po prostu podzbiorem zestawu Unicode); odrębny typ `bytes` reprezentuje surowe wartości bajtów (w tym media i zakodowany tekst); literały 2.x Unicode są obsługiwane w wersji 3.3 i nowszych w celu zapewnienia zgodności z wersją 2.x (są traktowane tak samo jak normalne ciągi znaków `str` z wersji 3.x):

```

>>> 'sp\xc4m'                                # 3.x: normalny ciąg str to tekst Unicode
'spÄm'
>>> b'a\x01c'                               # ciągi bytes to dane bajtowe
b'a\x01c'
>>> u'sp\u00c4m'                            # literały Unicode z wersji 2.x działają w
wersji 3.3+: traktowane są jak str
'spÄm'

```

W *Pythonie 2.x* normalne ciągi `str` obsługują zarówno 8-bitowe ciągi znaków (w tym tekst ASCII), jak i surowe wartości bajtów; odrębny typ `unicode` reprezentuje tekst Unicode; literały bajtowe 3.x są obsługiwane w wersji 2.6 i nowszych w celu zapewnienia zgodności z wersją 3.x (są traktowane tak samo jak normalne ciągi znaków `str` z wersji 2.x):

```

>>> print u'sp\xc4m'                         # 2.x: ciągi Unicode to osobny typ danych
spÄm
>>> 'a\x01c'                                 # Normalne ciągi str mogą zawierać tekst i
dane bajtowe
'a\x01c'
>>> b'a\x01c'                               # literały Unicode z wersji 3.x działają w
wersji 2.6+: traktowane są jak str
'a\x01c'

```

Formalnie zarówno w wersjach 2.x, jak i 3.x ciągi znaków innych niż Unicode są ciągami 8-bitowych bajtów, które są wyświetlane ze znakami ASCII, gdy jest to możliwe, a ciągi Unicode są ciągami kodów znaków Unicode, które przy zapisywaniu w plikach lub w pamięci mogą zajmować więcej niż jeden bajt. W rzeczywistości pojęcie bajtów nie dotyczy Unicode; niektóre sposoby kodowania do zapisania kodu znaku wymagają dwóch lub nawet czterech bajtów, a w niektórych schematach kodowania nawet prosty, 7-bitowy tekst ASCII nie jest zapisywany po jednym bajcie na znak:

```
>>> 'spam'                                # Znaki Unicode mogą zajmować 1, 2 lub nawet
4 bajty w pamięci

'spam'

>>> 'spam'.encode('utf8')                 # Ciąg znaków zakodowany w formacie UTF-8
zajmuje 4 bajty

b'spam'

>>> 'spam'.encode('utf16')                  # ale w standardzie UTF-16 zajmuje już 10
bajtów

b'\xff\xfe\x00p\x00a\x00m\x00'
```

Zarówno wersja 3.x, jak i 2.x obsługują również typ `bytearray`, z którym mieliśmy do czynienia już wcześniej, a który jest zasadniczo ciągiem bajtów (`str` w 2.x), obsługującym większość wewnętrznych operacji zmian obiektów listy.

Obie wersje, 3.x oraz 2.x, obsługują także kodowanie znaków spoza zestawu *ASCII* za pomocą szesnastkowych znaków ucieczki `\x` oraz krótkich `\u` i długich `\U` znaków Unicode, a także kodowanie całych plików deklarowane w plikach źródłowych programu. Poniżej przedstawiamy jako przykład wybrany znak spoza zestawu ASCII zakodowany na trzy sposoby w wersji 3.x (aby zobaczyć to samo w wersji 2.x, powinieneś dodać wiodący znak `u` oraz wywołanie funkcji `print`):

```
>>> 'sp\xc4\u00c4\U0000000c4m'
'spÄÄÄm'
```

Co te wartości oznaczają i jak są używane, zależy od tego, czy mówimy o ciągach znaków, czy ciągach bajtów. Ciagi znaków w wersji 3.x są zwykłymi ciągami znaków, a w wersji 2.x ciągami Unicode, natomiast ciągi bajtów w wersji 3.x są ciągami bajtów, a w wersji 2.x ciągami znaków. Wszystkie zaprezentowane sekwencje znaków ucieczki można wykorzystać do osadzania liczb całkowitych reprezentujących kody znaków Unicode w łańcuchach tekstu. Z kolei w ciągach bajtów do osadzania zakodowanego tekstu używamy tylko wartości szesnastkowych `\x`, a nie kodów znaków — zakodowane bajty mają takie same wartości jak kody znaków tylko w przypadku niektórych schematów kodowania i niektórych znaków:

```
>>> '\u00A3', '\u00A3'.encode('latin1'), b'\xA3'.decode('latin1')
('£', b'\xa3', '£')
```

Istotną różnicą jest to, że Python 2.x zezwala na mieszanie w wyrażeniach ciągów normalnych i Unicode, o ile normalny ciąg zawiera tylko znaki ASCII; dla kontrastu Python 3.x wykorzystuje bardziej ścisły model, który nigdy nie pozwala na mieszanie ciągów normalnych i bajtowych bez dokonania wyraźnej konwersji:

```
u'x' + b'y'                                # Działa w wersji 2.x (gdzie b jest opcjonalne i
ignorowane)

u'x' + 'y'                                  # Działa w wersji 2.x: u'xy'

u'x' + b'y'                                # Nie działa w wersji 3.3 (gdzie u jest
opcjonalne i ignorowane)
```

```

u'x' + 'y'                      # Działa w wersji 3.3: 'xy'
'x' + b'y'.decode()            # Działa w wersji 3.x, zamienia bajty na str:
'xy'
'x'.encode() + b'y'            # Działa w wersji 3.x, zamienia str na ciąg
bajtów: b'xy'

```

Poza przedstawionymi wyżej operacjami na łańcuchach przetwarzanie znaków Unicode ogranicza się głównie do przesyłania danych tekstowych do i z plików — tekst jest *kodowany* jako bajty, gdy jest przechowywany w pliku, i *dekodowany* do postaci znaków po ponownym załadowaniu do pamięci. Po załadowaniu zazwyczaj przetwarzamy tekst jako ciągi znaków tylko w postaci zdekodowanej.

Ze względu na taki model w wersji 3.x format plików jest specyficzny dla ich treści — *pliki tekstowe* implementują różne rodzaje kodowania oraz akceptują i zwracają ciągi *str*, ale *pliki binarne* zawierają surowe dane binarne w postaci ciągów bajtów. W Pythonie 2.x normalna zawartość plików to ciągi *str*, a dane typu *unicode* obsługiwane są przez specjalny moduł *codecs*.

Ze standardem Unicode spotkamy się jeszcze w dalszej części tego rozdziału przy okazji omawiania plików, ale resztę zagadnień zachowamy na nieco później. Kodowanie Unicode pojawi się na krótko w przykładzie z rozdziału 25. w połączeniu z symbolami walut, ale większość innych zagadnień odłożymy na bok do czasu, aż zostaną omówione zaawansowane tematy tej książki. Standard Unicode jest kluczowy w niektórych zastosowaniach, ale niemal każdy programista może sobie z tym poradzić, mając nawet tylko powierzchowną znajomość tematu. Jeżeli dane składają się tylko ze znaków ASCII, przetwarzanie ciągów znaków i plików jest bardzo do siebie zbliżone w obu wersjach, 2.X i 3.X. A jeżeli dopiero zaczynasz programować, możesz spokojnie pominąć większość szczegółów dotyczących standardu Unicode przynajmniej do czasu, dopóki nie opanujesz podstawowych zagadnień związanych z operacjami na ciągach znaków.

Dopasowywanie wzorców

Zanim przejdziemy dalej, warto odnotować, że żadna z metod łańcuchów znaków nie obsługuje przetwarzania tekstu opartego na wzorcach. Dopasowywanie wzorców tekstowych jest zaawansowanym narzędziem pozostającym poza zakresem niniejszej książki, jednak użytkownicy znający inne skryptowe języki programowania pewnie będą zainteresowani tym, że dopasowywanie wzorców w Pythonie odbywa się z wykorzystaniem modułu o nazwie *re*. Moduł ten zawiera odpowiednie metody dla wyszukiwania, dzielenia czy zastępowania tekstu, jednak dzięki użyciu wzorców do określania podłańcuchów znaków możemy to robić w znacznie bardziej elastyczny sposób.

```

>>> import re
>>> match = re.match('Witaj,[ \t]*(.*)Robinie', 'Witaj, sir Robinie')
>>> match.group(1)
'sir '

```

Powyższy przykład szuka podciągu znaków zaczynającego się od słowa *Witaj*, po którym następuje od zera do większej liczby tabulatorów lub spacji, dalej znajdują się dowolne znaki zapisane jako dopasowana grupa, a na końcu słowo *Robinie*. Kiedy taki podciąg zostanie odnaleziony, jego fragmenty dopasowane przez kolejne elementy wzorca umieszczone w nawiasach dostępne są jako grupy. Poniższy wzorzec wybiera na przykład trzy grupy rozdzielone znakami prawego ukośnika lub dwukropkami. Wynik takiej operacji jest podobny do podziału ciągu według zestawu alternatywnych separatorów, tak jak to pokazano poniżej:

```
>>> match = re.match('[/:](.*)[/:](.*)[/:](.*), '/usr/home:lumberjack')
```

```
>>> match.groups()
('usr', 'home', 'lumberjack')
>>> re.split('/:|', '/usr/home/lumberjack')
['', 'usr', 'home', 'lumberjack']
```

Dopasowywanie wzorców jest mocno zaawansowanym narzędziem do przetwarzania tekstu, ale w języku Python obsługiwane są jeszcze bardziej zaawansowane techniki przetwarzania tekstu i języków, takie jak analiza składniowa języków XML i HTML czy narzędzia do analizy języka naturalnego. Więcej przykładów dopasowywania wzorców tekstu i parsowania danych w formacie XML znajdziesz na końcu rozdziału 37., a teraz przejdziemy do omawiania kolejnego typu danych.

Listy

Obiekt reprezentujący listę jest w Pythonie najbardziej ogólnym rodzajem *sekwencji* dostępnej w tym języku. Listy to uporządkowane według kolejności zbiory obiektów dowolnego typu; nie mają one ustalonej wielkości. Listy są również *mutowalne* (ang. *mutable*) — w przeciwieństwie do łańcuchów znaków listy można modyfikować w miejscu, przypisując nowe wartości do wybranych indeksów, a także za pomocą wywołań odpowiednich metod, dzięki czemu stanowią one bardzo elastyczne narzędzie do reprezentowania dowolnych kolekcji, takich jak listy plików w folderze, zestawienia pracowników w firmie, wiadomości e-mail w skrzynce odbiorczej i tak dalej.

Operacje na typach sekwencyjnych

Ponieważ listy są sekwencjami, obsługują wszystkie operacje na typach sekwencyjnych przedstawione przy okazji omawiania łańcuchów znaków. Jedyną różnicą jest to, że wynikiem takich operacji zazwyczaj są listy, a nie łańcuchy znaków. Mając na przykład listę trzyelementową:

```
>>> L = [123, 'mielonka', 1.23]                      # Trzyelementowa lista z
elementami różnego typu
>>> len(L)                                         # Liczba elementów listy
3
```

możemy ją zaindeksować, wyodrębnić z niej wybrane elementy (wycinki listy) i wykonać pozostałe operacje zaprezentowane na przykładzie łańcuchów znaków:

```
>>> L[0]                                         # Indeksowanie według pozycji
123
>>> L[:-1]                                       # Wycinkiem listy jest nowa
lista
[123, 'mielonka']
>>> L + [4, 5, 6]                                 # Konkatenacje/powtórzenia także
zwracają nową listę
[123, 'mielonka', 1.23, 4, 5, 6]
>>> L * 2
```

```
[123, 'spam', 1.23, 123, 'spam', 1.23]  
>>> L  
zmienna L jest listą: [123, 'spam', 1.23, 123, 'spam', 1.23]  
# Oryginalna lista pozostaje bez zmian  
[123, 'mielonka', 1.23]
```

Operacje specyficzne dla typu

Listy w języku Python mogą nieco przypominać tablice znane z innych języków programowania, zazwyczaj jednak mają znacznie większe możliwości. Przede wszystkim nie mają żadnych ograniczeń odnośnie *typów danych* — przykładowo lista z poprzedniego przykładu zawiera trzy obiekty zupełnie różnych typów (liczbę całkowitą, łańcuch znaków i liczbę zmiennoprzecinkową). Co więcej, wielkość listy nie jest określana na stałe, co oznacza, że może ona się powiększać i zmniejszać w miarę potrzeb w rezultacie działania operacji specyficznych dla list.

```
>>> L.append('NI')  
obiektu na końcu listy  
>>> L  
[123, 'mielonka', 1.23, 'NI']  
>>> L.pop(2)  
obiektu ze środka listy  
1.23  
>>> L  
element z listy  
# "del L[2]" również usuwa  
[123, 'mielonka', 'NI']
```

Na powyższym listingu metoda `append` zwiększa rozmiar listy i wstawia na jej końcu nowy element. Metoda `pop` (lub jej odpowiednik, instrukcja `del`) usuwa element znajdujący się na pozycji o podanym indeksie, co sprawia, że lista się zmniejsza. Pozostałe metody list pozwalają na wstawianie elementu w dowolnym miejscu listy (`insert`) czy usuwanie elementu o podanej wartości (`remove`). Ponieważ listy są mutowalne, większość ich metod modyfikuje obiekty listy w miejscu, a nie tworzy nowy obiekt.

```
>>> M = ['bb', 'aa', 'cc']  
>>> M.sort()  
>>> M  
['aa', 'bb', 'cc']  
>>> M.reverse()  
>>> M  
['cc', 'bb', 'aa']
```

Zaprezentowana wyżej metoda listy o nazwie `sort` porządkuje listę w kolejności rosnącej, natomiast metoda `reverse` odwraca ją. W obu przypadkach lista modyfikowana jest w sposób bezpośredni.

Sprawdzanie granic

Choć listy nie mają ustalonej wielkości, Python nadal nie pozwala na odwoływanie się do elementów, które nie istnieją. Próba odwołania do elementu o nieistniejącym indeksie zawsze kończy się błędem, podobnie jak przypisywanie wartości do elementu spoza końca listy.

```
>>> L
[123, 'mielonka', 'NI']
>>> L[99]
...pominieto tekst błędu...
IndexError: list index out of range
>>> L[99] = 1
...pominieto tekst błędu...
IndexError: list assignment index out of range
```

Takie rozwiązanie jest celowe, ponieważ zazwyczaj błędem jest próba przypisania elementu poza końcem listy (jest to szczególnie paskudne w języku C, który nie sprawdza błędów w tak dużym stopniu jak Python). Zamiast po cichu powiększać listę, Python w takiej sytuacji zgłasza błąd. Aby powiększyć listę, powinieneś zamiast tego wywołać odpowiednią metodę, taką jak `append`.

Zagnieżdżanie

Jedną z fajnych właściwości podstawowych typów danych w Pythonie jest obsługa dowolnego *zagnieżdżania* — możemy je zagnieżdżać w dowolnej kombinacji i na dowolną głębokość (czyli można na przykład utworzyć listę zawierającą słownik, który z kolei zawiera kolejną listę). Bezpośrednim zastosowaniem tej właściwości jest reprezentowanie w Pythonie macierzy, czy inaczej mówiąc, tablic wielowymiarowych. Lista mieszcząca zagnieżdżone listy doskonale się sprawdzi w prostych zastosowaniach (w niektórych interfejsach w wierszach 2. i 3. pojawią się znaki kontynuacji wiersza (...), ale w środowisku IDLE tak się nie dzieje):

```
>>> M = [[1, 2, 3], # Macierz 3x3 w postaci list
          zagnieżdżonych
          [4, 5, 6], # Kod może się rozciągać na kilka wierszy,
          jeśli znajduje się w nawiasach
          [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Powyżej utworzyliśmy listę zawierającą trzy inne listy i w rezultacie otrzymujemy reprezentację macierzy liczbowej 3×3 . Dostęp do takiej struktury można uzyskać na wiele sposobów:

```
>>> M[1] # Pobranie drugiego wiersza
[4, 5, 6]
>>> M[1][2] # Pobranie drugiego wiersza, a z niego
              trzeciego elementu
6
```

Pierwsza operacja pobiera cały drugi wiersz, natomiast druga pobiera trzeci element z tego wiersza (operacja jest wykonywana od lewej do prawej, podobnie jak to było w przypadku

dzielenia ciągów znaków i wyodrębniania z nich określonych elementów). Łączenie ze sobą indeksów wciąża nas coraz głębiej i głębiej w zagnieżdżoną strukturę obiektu [2].

Listy składane

Poza operacjami na typach sekwencyjnych i metodami obiektów listy Python obsługuje bardziej zaawansowane operacje, znane pod nazwą *wyrażeń list składanych* (ang. *list comprehension expressions*; inaczej *wyrażenia listowe*), które świetnie nadają się do przetwarzania struktur takich jak nasza macierz. Przypuśćmy na przykład, że chcemy pobrać drugą kolumnę naszej prostej macierzy. Za pomocą zwykłego indeksowania łatwo jest pobierać wiersze, ponieważ macierze przechowywane są wierszami. Podobnie łatwo jest pobrać kolumnę za pomocą wyrażenia listy składanej:

```
>>> col2 = [row[1] for row in M] # Zebranie elementów z drugiej kolumny
>>> col2
[2, 5, 8]
>>> M # Macierz pozostaje bez zmian
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Listy składane pochodzą z notacji zbiorów. Służą do budowania nowych list poprzez wykonanie wyrażenia na każdym elemencie sekwencji po kolej, jeden po drugim, od lewej strony do prawej. Wyrażenia listy składanej zapisywane są w nawiasach kwadratowych (by podkreślić fakt, że tworzą listę) i składają się z wyrażenia i konstrukcji pętli, które dzielą ze sobą nazwę zmiennej (tutaj `row`). Powyższe wyrażenie listy składanej oznacza mniej więcej tyle: „Zwróć `row[1]` dla każdego wiersza w macierzy `M` w nowej liście”. Wynikiem jest nowa lista zawierająca drugą kolumnę macierzy.

Oczywiście wyrażenia listy składanej mogą w praktyce być bardziej złożone:

```
>>> [row[1] + 1 for row in M] # Dodanie 1 do każdego elementu w drugiej kolumnie
[3, 6, 9]
>>> [row[1] for row in M if row[1] % 2 == 0] # Odfiltrowanie elementów nieparzystych
[2, 8]
```

Pierwsza operacja z listingu powyżej dodaje 1 do każdego pobranego elementu, natomiast druga wykorzystuje wyrażenie `if` do odfiltrowania liczb nieparzystych z wyniku, używając do tego reszty z dzielenia (wyrażenie z operatorem `%`). Wyrażenia listowe tworzą nowe listy wyników, ale można ich również używać do przetwarzania w pętli dowolnych obiektów *iterowalnych* (ang. *iterable objects*), o których będziemy mówić w dalszej części tego rozdziału. W przykładzie poniżej pokazujemy zastosowanie list składanych do przetwarzania zakodowanej listy współrzędnych oraz łańcucha znaków:

```
>>> diag = [M[i][i] for i in [0, 1, 2]] # Pobranie przekątnej z macierzy
>>> diag
[1, 5, 9]
>>> doubles = [c * 2 for c in 'mielonka'] # Powtórzenie znaków w łańcuchu
>>> doubles
```

```
['mm', 'ii', 'ee', 'll', 'oo', 'nn', 'kk', 'aa']
```

Wyrażen list składanych można również używać do zbierania wielu wartości, o ile będziemy je umieszczać w zagnieżdżonej kolekcji. Poniższy przykład ilustruje użycie wbudowanej funkcji `range`, która generuje kolejne liczby całkowite; pamiętaj, że w wersji 3.x do utworzenia i wyświetlenia listy musisz użyć funkcji `list` (w wersji 2.x lista tworzona i wyświetlana jest od razu):

```
>>> list(range(4))                                # 0..3 (w wersji 3.x wymagane jest
użycie funkcji list())
[0, 1, 2, 3]
>>> list(range(-6, 7, 2))                         # -6 do +6 z krokiem 2 (w wersji
3.x musimy użyć funkcji list())
[-6, -4, -2, 0, 2, 4, 6]
>>> [[x ** 2, x ** 3] for x in range(4)]          # Wiele wartości, filtrowanie z
użyciem polecenia if
[[0, 0], [1, 1], [4, 8], [9, 27]]
>>> [[x, x / 2, x * 2] for x in range(-6, 7, 2) if x > 0]
[[2, 1, 4], [4, 2, 8], [6, 3, 12]]
```

Jak zapewne zauważyleś, wyrażenia listowe i powiązane z nimi zagadnienia, takie jak wbudowane funkcje `map` oraz `filter`, są trochę zbyt zaawansowane, by omawiać je bardziej szczegółowo już teraz. Najważniejsze w tym krótkim wprowadzeniu jest to, abyś wiedział, że w swoim arsenale Python posiada zarówno narzędzia podstawowe, jak i bardziej zaawansowane. Listy składane są elementem opcjonalnym, jednak w praktyce często okazują się bardzo przydatne i nierzadko pozwalają znakomicie zwiększyć szybkość przetwarzania danych. Wyrażenia listowe działają na wszystkich sekwencyjnych typach danych, a także na pewnych typach, które sekwencjami nie są. Wróćmy do nich w dalszej części książki.

Warto zauważyć, że w najnowszych wersjach Pythona uogólniono i ujednolicono składnię wyrażeń listowych, dostosując ją również do innych zastosowań; obecnie nie są używane wyłącznie do tworzenia list. Na przykład umieszczenie takiego wyrażenia w *nawiasach* można wykorzystać do tworzenia *generatorów*, które podają wyniki na żądanie. Aby to zilustrować, użyjemy wbudowanej funkcji `sum` do sumowania elementów w sekwencji — w przykładzie poniżej będziemy na żądanie sumować wszystkie elementy w wierszach naszej macierzy:

```
>>> G = (sum(row) for row in M)                  # Utworzenie generatora sum wierszy
>>> next(G)                                      # użycie funkcji iter(G) nie jest tu
wymagane
6
>>> next(G)                                      # Wykonanie iteracji, obliczenie
kolejnej sumy i wyświetlenie wyniku
15
>>> next(G)
24
```

Wbudowana funkcja `map` może wykonać podobne zadanie, pobierając jako parametry wywołania funkcję oraz listę i zwracając nową listę, która powstaje w wyniku wywołania funkcji przekazanej w pierwszym parametrze dla każdego elementu listy przekazanej w drugim parametrze. Podobnie jak to miało miejsce w przypadku funkcji `range`, w wersji 3.x do

wyświetlenia otrzymanych wartości konieczne jest wywołanie funkcji `map` jako parametru funkcji `list`; nie jest to potrzebne w wersji 2.x, w której funkcja `map` tworzy listę wszystkich wyników naraz. Nie jest również konieczne w kilku innych kontekstach, gdzie iteracje odbywają się automatycznie, chyba że wymagane jest również wielokrotne skanowanie lub zachowania podobne do listy:

```
>>> list(map(sum, M))                                # Wykonanie funkcji sum dla
elementów macierzy M
[6, 15, 24]
```

W wersjach 2.x i 3.x Pythona składnię wyrażeń listowych można także wykorzystać do tworzenia *zbiorów* oraz *słowników*:

```
>>> {sum(row) for row in M}                          # Utworzenie zbioru sum wierszy
{24, 6, 15}
>>> {i : sum(M[i]) for i in range(3)}            # Utworzenie tabeli klucz-wartość
sum wierszy
{0: 6, 1: 15, 2: 24}
```

Tak naprawdę listy, zbiory, słowniki oraz generatory można tworzyć za pomocą wyrażeń listowych zarówno w wersji 2.x, jak i 3.x:

```
>>> [ord(x) for x in 'mieloonka']                 # Lista numerów porządkowych
znaków
[109, 105, 101, 108, 111, 111, 110, 107, 97]
>>> {ord(x) for x in 'mieloonka'}                  # Zbiory pozwalają na usunięcie
duplikatów
{97, 101, 105, 107, 108, 109, 110, 111}
>>> {x: ord(x) for x in 'mieloonka'}              # Klucze słownika są unikalne
{'a': 97, 'e': 101, 'i': 105, 'k': 107, 'm': 109, 'l': 108, 'o': 111, 'n':
110}
>>> (ord(x) for x in 'spaam')                      # Generator wartości
<generator object <genexpr> at 0x000000000254DAB0>
```

Aby dobrze zrozumieć obiekty takie jak generatory, zbiory oraz słowniki, musimy przejść nieco dalej.

Słowniki

Słowniki w Pythonie to coś zupełnie innego [\[3\]](#) — nie są sekwencjami, ale są za to określane jako *odwzorowania* (ang. *mappings*). Odwzorowania są również zbiorami innych obiektów, jednak w słownikach obiekty są przechowywane według klucza, a nie ich pozycji względnej. W odwzorowaniach nie możemy mówić o żadnej kolejności elementów od lewej do prawej strony itp.; zamiast tego mamy po prostu zestawy kluczy powiązanych z odpowiednimi wartościami. Słowniki, jedyny typ odwzorowania w zbiorze typów podstawowych Pythona, są również *mutowalne* — mogą być modyfikowane w miejscu i podobnie do list, mogą zwiększać i zmniejszać się na żądanie. Podobnie jak listy, są elastycznym narzędziem do reprezentowania kolekcji, ale ich bardziej *mnemoniczne* klucze lepiej nadają się do zastosowania, kiedy elementy

kolekcji są nazwane lub oznaczone, na przykład jak to ma miejsce w przypadku pól rekordów bazy danych.

Operacje na odwzorowaniach

Kiedy słowniki zapiszemy w formie literalów, zostają umieszczone w nawiasach klamrowych i składają się z serii par *klucz: wartość*. Słowniki są przydatne w sytuacjach, gdy chcemy powiązać zbiór wartości z kluczami — na przykład opisać właściwości czegoś. Rozważmy poniższy słownik składający się z trzech elementów (których kluczami będą jedzenie, ilość i kolor; mogą to być na przykład szczegóły jakiegoś hipotetycznego menu):

```
>>> D = {'jedzenie': 'Mielonka', 'ilość': 4, 'kolor': 'różowy'}
```

Możemy teraz poindeksować ten słownik według kluczy, aby pobierać i zmieniać powiązane z nimi wartości. Operacja indeksowania słownika używa tej samej składni, co w przypadku sekwencji, jednak elementem znajdującym się w nawiasach kwadratowych jest klucz, a nie pozycja względna:

```
>>> D['jedzenie'] # Pobranie wartości klucza  
"jedzenie"  
'Mielonka'  
>>> D['ilość'] += 1 # Dodanie 1 do wartości klucza  
"ilość"  
>>> D  
{'jedzenie': 'Mielonka', 'kolor': 'różowy', 'ilość': 5}
```

Choć forma literala z nawiasami klamrowymi jest czasami używana, znacznie częściej możesz spotkać słowniki tworzone w inny sposób (w praktyce bardzo rzadko zdarza się przecież, że znasz wszystkie dane przetwarzane przez Twój program przed jego uruchomieniem). Poniższy przykład rozpoczyna się od utworzenia pustego słownika, który jest następnie wypełniany po jednym kluczu na raz. W przeciwieństwie do przypisywania wartości do elementów spoza listy, które nie są dozwolone, przypisania do nowych, nieistniejących jeszcze kluczy słownika powodują utworzenie takich kluczy.

```
>>> D = {}  
>>> D['imię'] = 'Robert' # Tworzenie kluczy przez  
przypisanie  
>>> D['zawód'] = 'programista'  
>>> D['wiek'] = 40  
>>> D  
{'wiek': 40, 'zawód': 'programista', 'imię': 'Robert'}  
>>> print(D['imię'])  
Robert
```

Powyżej wykorzystaliśmy klucze jako nazwy pól w spisie opisującym wybraną osobę. W innych zastosowaniach słowniki mogą być również wykorzystywane do zastępowania operacji wyszukiwania — poindeksowanie słownika według klucza jest często najszybszą metodą zakodowania wyszukiwania w Pythonie.

Jak się przekonasz nieco później, możemy również tworzyć słowniki, przekazując do typu `dict` odpowiednie zestawy argumentów (specjalna składnia `nazwa = wartość` w wywołaniach funkcji)

albo wyniki połączenia ze sobą sekwencji kluczy i wartości uzyskanych w czasie działania programu (np. z plików). Oba przykłady pokazane poniżej tworzą ten sam słownik co w poprzednim przykładzie, choć pierwsza wersja wymaga nieco mniejszej ilości pisania:

```
>>> bob1 = dict(imię='Robert', zawód='programista', wiek=40)          #
Słowa kluczowe

>>> bob1

{'wiek': 40, 'imię': 'Robert', 'zawód': 'programista'}

>>> bob2 = dict(zip(['imię', 'zawód', 'wiek'], ['Robert', 'programista',
40])) # Łączenie

>>> bob2

{'zawód': 'programista', 'imię': 'Robert', 'wiek': 40}
```

Zwróć uwagę, że kolejność wyświetlanych kluczy słownika w obu przykładach jest różna. Odwzorowania nie są uporządkowane w żadnej określonej kolejności, więc jeżeli nie dopisze Ci szczęście, będą wyświetlane w innej kolejności, niż je wpisałeś. Kolejność elementów odwzorowania może się różnić w zależności od wersji Pythona, ale nie powinieneś na niej polegać i oczekwać, że Twoje otrzymane wyniki będą pasowały do tego, co zobaczyłeś w tej książce.

Zagnieżdżanie raz jeszcze

W poprzednim przykładzie wykorzystaliśmy słownik do opisania hipotetycznej osoby za pomocą trzech kluczy. Założymy jednak, że informacje są bardziej złożone. Być może konieczne będzie zanotowanie imienia i nazwiska, a także nazw wykonywanych zawodów czy posiadanych tytułów. Taki scenariusz w praktyce pozwoli nam zapoznać się z kolejnym zastosowaniem mechanizmu zagnieżdżania obiektów. Poniższy słownik, zapisany w całości w postaci jednego literału, zawiera bardziej ustrukturyzowane informacje:

```
>>> rec = {'dane osobowe': {'imię': 'Robert', 'nazwisko': 'Zielony'},
           'zawód': ['programista', 'inżynier'],
           'wiek': 40.5}
```

W powyższym przykładzie znowu mamy słownik z trzema kluczami (dane osobowe, zawód oraz wiek), natomiast ich wartości stają się nieco bardziej skomplikowane. Zagnieżdżony słownik z danymi osobowymi może pomieścić kilka informacji, natomiast zagnieżdżona lista z zawodem mieści kilka ról i można ją w przyszłości rozszerzyć. Dostęp do elementów tej struktury można uzyskać w podobny sposób jak w przypadku pokazanej wcześniej macierzy, jednak tym razem indeksami będą klucze słownika, a nie wartości przesunięcia listy.

```
>>> rec['dane osobowe']                                # 'dane osobowe' to zagnieżdżony
słownik

{'nazwisko': 'Zielony', 'imię': 'Robert'}

>>> rec['dane osobowe']['nazwisko']                  # Indeksowanie zagnieżdzonego
słownika

'Zielony'

>>> rec['zawód']                                     # 'zawód' to zagnieżdżona lista

['programista', 'inżynier']
```

```

>>> rec['zawód'][-1]                                # Indeksowanie zagnieżdżonej
listy
'inżynier'

>>> rec['zawód'].append('leśniczy')               # Rozszerzenie listy zawodów
Roberta

>>> rec
{'wiek': 40.5, 'zawód': ['programista', 'inżynier', 'leśniczy'], 'dane
osobowe': {'nazwisko': 'Zielony', 'imię': 'Robert'}}

```

Warto zwrócić uwagę na to, jak ostatnia operacja rozszerza osadzoną listę zawodów. Ponieważ lista zawodów jest fragmentem pamięci oddzielnym od zawierającego ją słownika, może dowolnie rosnąć i kurczyć się (rozkład pamięci obiektów omówiony zostanie w dalszej części książki).

Prawdziwym powodem pokazania tego przykładu jest chęć zademonstrowania *elastyczności* podstawowych typów obiektów Pythona. Jak widać, zagnieżdżanie pozwala na budowanie skomplikowanych struktur informacji w sposób łatwy i bezpośredni. Zbudowanie podobnej struktury w języku niskiego poziomu, takim jak C, byłoby zmuśne i wymagałoby o wiele większej ilości kodu. Musielibyśmy zaprojektować i zadeklarować układ struktury i tablic, wypełnić je wartościami i wreszcie połączyć wszystko ze sobą. W Pythonie wszystko to dzieje się automatycznie — jedno wyrażenie tworzy za nas całą zagnieżdżoną strukturę obiektów. Tak naprawdę jest to jedna z najważniejszych zalet języków skryptowych, takich jak Python.

Co równie ważne, w języku niższego poziomu musielibyśmy uważnie czyścić przestrzeń zajmowaną przez obiekty, które jej już dłużej nie potrzebują. W Pythonie, kiedy znika ostatnie odwołanie do obiektu (na przykład gdy do zmiennej zostanie przypisane coś innego), całe miejsce w pamięci zajmowane przez tę strukturę obiektu jest automatycznie zwalniane.

```

>>> rec = 0                                         # Miejsce zajmowane przez obiekt
zostaje odzyskane

```

Z technicznego punktu widzenia Python korzysta z mechanizmu znanego jest pod nazwą *czyszczenia pamięci* (ang. *garbage collection*). Nieużywana pamięć jest czyszczona w miarę wykonywania programu, co zwalnia nas z odpowiedzialności za zarządzanie takimi szczegółami w naszym kodzie. W standardowych implementacjach Pythona (określanych jako CPython) zajmowana pamięć jest odzyskiwana natychmiast, kiedy tylko zniknie ostatnie odwołanie do obiektu. W rozdziale 6. omówimy tę kwestię bardziej szczegółowo, a na razie powinna nam wystarczyć wiedza, że możemy swobodnie korzystać z obiektów, bez konieczności troszczenia się o alokowanie dla nich miejsca w pamięci i zwalnianie go.

Strukturę rekordów podobną do przedstawionej powyżej znajdziesz również w rozdziale 8., rozdziale 9. i rozdziale 27., gdzie użyjemy jej do porównywania i zestawiania list, słowników, krotek, krotek nazwanych i klas, czyli całego szeregu struktur danych, które bardziej szczegółowo omówimy w dalszej części książki^[4].

Brakujące klucze – testowanie za pomocą if

Ze względu na fakt, że słowniki są odwzorowaniami, obsługują dostęp do poszczególnych elementów tylko według kluczy. Ponadto obsługują również operacje specyficzne dla poszczególnych typów obiektów za pomocą wywołań odpowiednich *metod*, które są przydatne w wielu typowych przypadkach użycia. Na przykład, choć możemy rozszerzać słowniki, przypisując nowe wartości do nowych, nieistniejących kluczy, to już próba pobrania wartości z nieistniejącego klucza zakończy się błędem:

```

>>> d = {'a': 1, 'b': 2, 'c': 3 }

```

```

>>> D
{'a': 1, 'c': 3, 'b': 2 }

>>> D['e'] = 99                                # Przypisanie nowego klucza
rozszerza słownik

>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                                    # Odwołanie do nieistniejącego
klucza jest błędem
...pominieto tekst błędu...

KeyError: 'f'

```

Tego właśnie oczekujemy — zazwyczaj pobieranie czegoś, co nie istnieje, jest błędem programistycznym. Jednak w niektórych programach nie zawsze będziemy w momencie pisania kodu wiedzieć, jakie klucze będą dostępne. W jaki sposób można sobie poradzić w takim przypadku i uniknąć błędów? Jednym z rozwiązań jest sprawdzenie tego zawsze. Wyrażenie `in` słownika pozwala na sprawdzenie istnienia klucza i odpowiednie zachowanie w zależności od wyniku tego sprawdzenia dzięki instrukcji `if`. W przykładzie poniżej naciśnij dwukrotnie klawisz `Enter`, aby po wpisaniu kodu uruchomić interaktywnie `if` (jak wyjaśniono w rozdziale 3., pusty wiersz w interaktywnym monicie oznacza „`idź`”); podobnie jak w przypadku wcześniejszych przykładów ze słownikami i listami wielowierszowymi, w niektórych interfejsach znak zachęty zmienia się na „...” dla drugiego i dalszych wierszy.

```

>>> 'f' in D
False
>>> if not 'f' in D:                         # wyrażenie sprawdzające
                                               
                                                print('nie istnieje')
nie istnieje

```

Na temat instrukcji `if` powiemy więcej w kolejnych rozdziałach, ale forma, której tutaj używamy, jest bardzo prosta: składa się ze słowa kluczowego `if`, po nim następuje wyrażenie, którego wynik interpretowany jest jako prawdziwy lub fałszywy, a następnie blok kodu do wykonania, jeśli wyrażenie będzie prawdziwe. W pełnej formie instrukcja `if` może również zawierać klauzulę `else` dla przypadku domyślnego oraz jedną lub więcej klauzuł `elif` („`else if`”) sprawdzających inne testy. Jest to podstawowa instrukcja `wyboru` w Pythonie, która wraz z jej trójskładnikowym kuzynem `if/else` (którego spotkamy za chwilę) oraz filtrem `if`, stosowanym w wyrażeniach listowych, jest używana do kodowania logiki wyboru i decyzji w naszych skryptach.

Jeżeli w przeszłości używałeś innych języków programowania, możesz zastanawiać się, skąd Python wie, kiedy kończy się instrukcja `if`. Reguły składni Pythona wyjaśnimy dogłębnie w dalszych rozdziałach, ale w skrócie, jeżeli masz więcej niż jedną akcję do uruchomienia w bloku instrukcji, po prostu używasz takiego samego wcięcia dla wszystkich instrukcji tego bloku — takie rozwiązanie powoduje, że kod programu staje się bardziej czytelny i zmniejsza się liczba znaków, które musisz wpisać:

```

>>> if not 'f' in D:
                                               
                                                print('nie istnieje')
                                               
                                                print('naprawdę nie istnieje...')    # wyrażenia w bloku mają takie
samo wcięcie

```

```
nie istnieje  
naprawdę nie istnieje...
```

Oprócz testu z operatorem `in` istnieje wiele innych sposobów unikania próby dostępu do nieistniejących kluczy w tworzonych przez nas słownikach: metoda `get`, indeks warunkowy z wartością domyślną; metoda `has_key` w Pythonie 2.x (podobna do operatora `in`, nie jest już dostępna w wersji 3.x); instrukcja `try`, czyli narzędzie, które poznamy po raz pierwszy w rozdziale 10., pozwalająca na przechwytywanie i obsługę wyjątków; wyrażenie trójskładnikowe `if/else`, które jest w gruncie rzeczy instrukcją `if` umieszczoną w jednym wierszu. Oto kilka przykładów takich poleceń:

```
>>> value = D.get('x', 0)                                # Indeks, ale z wartością  
domyślną  
>>> value  
  
0  
>>> value = D['x'] if 'x' in D else 0                  # Forma wyrażenia if/else  
>>> value  
  
0
```

O szczegółach takich poleceń opowiadamy już w kolejnym rozdziale, a teraz przejdziemy do omawiania kolejnej metody słowników i jej typowych zastosowań.

Sortowanie kluczy — pętle for

Jak wspomniano wcześniej, ponieważ słowniki nie są sekwencjami, nie zachowują żadnej uporządkowanej kolejności elementów. Oznacza to, że jeżeli utworzymy słownik i wyświetlimy jego zawartość, klucze mogą zostać zwrócone w innej kolejności, niż je wpisywaliśmy (kolejność może się różnić także w zależności od wersji Pythona i kilku innych czynników):

```
>>> D = {'a': 1, 'b': 2, 'c': 3}  
>>> D  
{'a': 1, 'c': 3, 'b': 2}
```

Co jednak można zrobić, kiedy potrzebne nam jest wymuszenie określonej kolejności elementów słownika? Jednym z często spotykanych rozwiązań jest tu pobranie ze słownika listy kluczy za pomocą metody `keys`, posortowanie tej listy za pomocą metody `sort`, a następnie wyświetlenie wyników za pomocą pętli `for` (podobnie jak w przypadku polecenia `if`, powinieneś pamiętać o dwukrotnym naciśnięciu klawisza `Enter` po utworzeniu kodu pętli przedstawionego poniżej oraz o pominięciu zewnętrznych nawiasów, jeżeli korzystasz z Pythona 2.x i używasz funkcji `print`).

```
>>> Ks = list(D.keys())                                # Nieuporządkowana lista kluczy  
>>> Ks  
użyj list()  
['a', 'c', 'b']  
>>> Ks.sort()                                         # Posortowana lista kluczy  
>>> Ks  
['a', 'b', 'c']
```

```
>>> for key in Ks:                                # Iteracja przez posortowane
    klucze

        print(key, '=>', D[key])                  # <== Tutaj należy dwukrotnie
nacisnąć Enter

a => 1
b => 2
c => 3
```

Ten proces składa się jednak z trzech etapów, natomiast jak zobaczymy w kolejnych rozdziałach, w nowszych wersjach Pythona można go wykonać w jednym kroku — dzięki nowej, wbudowanej funkcji o nazwie `sorted`. Wywołanie funkcji `sorted` zwraca wynik i sortuje różne typy obiektów; w tym przypadku automatycznie sortuje klucze słownika.

```
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for key in sorted(D):
    print(key, '=>', D[key])

a => 1
b => 2
c => 3
```

Poza zaprezentowaniem słowników powyższy przypadek posłużył nam jako pretekst do wprowadzenia pętli `for` dostępnej w Pythonie. Pętla `for` jest prostym i wydajnym sposobem przechodzenia przez wszystkie elementy sekwencji i wykonania bloku kodu dla każdego z nich po kolei. Zdefiniowana przez użytkownika zmienna pętli (tutaj: `key`) wykorzystana jest do odwoływania się do aktualnie przetwarzanego elementu. Wynikiem działania kodu przedstawionego w naszym przykładzie jest wyświetlenie par kluczy i wartości słownika w kolejności posortowanej według kluczy.

Pętla `for`, a także jej bardziej ogólna wersja, pętla `while`, są podstawowymi sposobami kodowania *powtarzalnych* zadań w skryptach. Tak naprawdę jednak pętla `for`, a także jej krewniacy, listy składane (mówiliśmy o nich już wcześniej), to operacje wykonywane na sekwencjach, które zadziałają na każdym obiekcie będącym sekwencją, a nawet, podobnie jak to miało miejsce w przypadku list składanych, na pewnych obiektach, które sekwencjami nie są. Poniżej zamieszczamy przykład zastosowania pętli `for` do przechodzenia przez kolejne znaki w łańcuchu i wyświetlania ich na ekranie przy użyciu wielkich liter:

```
>>> for c in 'mielonka':
    print(c.upper())

M
I
E
L
O
N
```

K

A

Pętla `while` w języku Python jest narzędziem bardziej ogólnym i nie jest ograniczona do przetwarzania sekwencji, ale generalnie jej implementacja wymaga wpisania nieco większej ilości kodu:

```
>>> x = 4
>>> while x > 0:
    print('mielonka!' * x)
    x -= 1
mielonka!mielonka!mielonka!mielonka!
mielonka!mielonka!mielonka!
mielonka!mielonka!
mielonka!
```

Instrukcje, składnia i narzędzia powiązane z pętlami omówione zostaną bardziej szczegółowo w dalszej części książki. Zanim przejdziemy dalej, muszę jednak przyznać, że ta sekcja nie była tak wyczerpująca, jak moglibyśmy ją zrobić. W rzeczywistości pętla `for` i jej wszystkie pochodne, które przechodzą kolejno przez obiekty od lewej do prawej, nie są tylko operacjami *sekwencyjnymi*, są przede wszystkim operacjami *iteracyjnymi*, o których opowiem bardziej szczegółowo już w kolejnym podrozdziale.

Iteracja i optymalizacja

Jeżeli pętla `for` z poprzedniego podrozdziału przypomina Ci nieco omawiane wcześniej wyrażenia z listami składanymi, to tak właśnie powinno być — oba narzędzia są ogólnymi narzędziami iteracyjnymi. Tak naprawdę oba rozwiązania będą działały na każdym iterowalnym obiekcie zgodnym z protokołem iteracji, czyli rozpoznawaną w Pythonie koncepcją, która leży u podstaw wszystkich narzędzi iteracyjnych tego języka.

Mówiąc w skrócie, obiekt jest *iterowalny*, jeżeli jest albo fizycznie zapisaną sekwencją w pamięci, albo obiektem, który w kontekście iteracji generuje jeden element na raz, tworząc rodzaj „wirtualnej” sekwencji. Mówiąc bardziej formalnie, oba typy obiektów są uważane za iterowalne, ponieważ obsługują *protokół iteracji* — odpowiadają na wywołanie iteracji obiektem, który po wywołaniu funkcji `next` przechodzi do kolejnego elementu i generuje wyjątek po dotarciu do końca sekwencji.

Wyrażenia składane tworzące *generatory*, które pokazywaliśmy wcześniej, są takimi właśnie obiektami: ich kolejne wartości nie są zapisywane w pamięci, ale są tworzone w odpowiedzi na kolejne żądania, zwykle za pomocą narzędzi iteracyjnych. W języku Python *obiekty plików* w podobny sposób przechodzą przez zawartość wiersz po wierszu, gdy są używane przez narzędzie iteracyjne: zawartość pliku nie znajduje się na liście, ale jest pobierana na żądanie. Generatory i obiekty plików są zatem w Pythonie obiektami iterowalnymi — jest to kategoria obiektów, która w wersji 3.x została rozszerzona tak, aby objąć podstawowe narzędzia, takie jak `range` i `map`. Dzięki możliwości dostarczania wyników na żądanie narzędzia te mogą zarówno oszczędzać pamięć, jak i zwiększać wydajność przetwarzania.

Na temat protokołu iteracji będę miał więcej do powiedzenia w dalszej części książki. Na razie powinieneś zapamiętać, że każde narzędzie Pythona, które przegląda obiekty od lewej do prawej strony, wykorzystuje protokół iteracji. Dlatego właśnie wywołanie funkcji `sorted`, wykorzystane wyżej, działa bezpośrednio na słowniku. Nie musimy wywoływać metody `keys` w

celu otrzymania sekwencji, ponieważ słowniki poddają się iteracji, a funkcja `next` zwraca w ich przypadku kolejne klucze.

Oznacza to także, że każde wyrażenie list składanych — jak poniższe, obliczające kwadrat z listy liczb:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]  
>>> squares  
[1, 4, 9, 16, 25]
```

zawsze może zostać zapisane w kodzie jako odpowiednik pętli `for` tworzącej listę ręcznie — poprzez dodanie do niej kolejnych przechodzonych elementów:

```
>>> squares = []  
>>> for x in [1, 2, 3, 4, 5]:                      # Taką operację wykonuje lista  
    składana  
        squares.append(x ** 2)                      # Wewnętrznie wykorzystuje  
protokół iteracji  
>>> squares  
[1, 4, 9, 16, 25]
```

Oba narzędzia wykorzystują wewnętrznie protokół iteracji i dają ten sam wynik. Lista składana, a także powiązane z nią narzędzia programowania funkcyjonalnego, takie jak `map` oraz `filter`, będą dla niektórych rodzajów kodu działały szybciej od pętli `for` (być może nawet dwa razy szybciej) — ta cecha może mieć znaczenie w przypadku większych zbiorów danych. Mimo to warto podkreślić, że pomiar wydajności może być w Pythonie dość złożonym zagadnieniem, ponieważ duża część kodu jest wewnętrznie optymalizowana, a wydajność Pythona może się zmieniać z wydania na wydanie.

Najważniejszą regułą jest w Pythonie kodowanie w sposób prosty oraz czytelny i martwienie się o wydajność dopiero później, kiedy sam program już działa i przekonamy się, że wydajność jest dla nas rzeczywistym problemem. Najczęściej kod będzie wystarczająco szybki. Jeżeli musisz w nim trochę pomajstrować pod kątem wydajności, Python posiada narzędzia, które mogą Ci w tym pomóc, w tym moduły `time` i `timeit` służące do pomiaru prędkości działania fragmentów kodu oraz moduł `profile` wspomagający wykrywanie i identyfikowanie wąskich gardeł w działaniu programu.

Więcej szczegółowych informacji na ten temat znajdziesz w dalszej części książki (zobacz zwłaszcza studium przypadku testów wydajności w rozdziale 21.) oraz w dokumentacji Pythona, a teraz przejdziemy do omawiania kolejnego typu danych.

Krotki

Obiekt *krotki* (ang. *tuple*) można w przybliżeniu opisać jako listę, której nie można zmodyfikować — krotki są *sekwencjami*, podobnie jak listy, jednak są też niemutowalne (ang. *immutable*), tak jak łańcuchy znaków. Funkcjonalnie rzecz biorąc, krotki służą do reprezentowania stałych kolekcji przedmiotów, na przykład elementów powiązanych z określoną datą. Składniowo są zwykle zapisywane w nawiasach okrągłych, a nie kwadratowych i obsługują dowolne typy, dowolne zagnieżdżanie i standardowe operacje na sekwencjach:

```
>>> T = (1, 2, 3, 4)                      # Krotka z 4 elementami  
>>> len(T)                                # Długość krotki
```

```
4
```

```
>>> T + (5, 6) # Konkatenacja  
(1, 2, 3, 4, 5, 6)  
>>> T[0] # Indeksowanie i wycinki  
1
```

Krotki posiadają również metody specyficzne dla tego typu (przynajmniej w Pythonie 2.6 i 3.0), ale nie jest ich tak wiele jak w przypadku list:

```
>>> T.index(4) # Metody krotek – wartość 4  
znajduje się na 3. pozycji  
3  
>>> T.count(4) # Wartość 4 w tej krotce pojawia  
się tylko raz  
1
```

Podstawową cechą wyróżniającą krotki jest to, że po utworzeniu nie można ich zmodyfikować. Oznacza to, że są sekwencjami niemutowalnymi (krotki zawierające jeden element wymagają dodania na końcu przecinka):

```
>>> T[0] = 2 # Krotki są niemutowalne  
...pominieto tekst błędu...  
TypeError: 'tuple' object does not support item assignment  
>>> T = (2,) + T[1:] # Utwórz nową krotkę dla nowej  
wartości  
>>> T  
(2, 2, 3, 4)
```

Podobnie jak listy i słowniki, krotki obsługują typy mieszane i zagnieżdżanie, ale nie mogą rosnąć, ani się kurczyć, ponieważ są niemutowalne (nawiasy otaczające elementy krotki można często pominąć, tak jak w przykładzie poniżej; to przecinki są tym, co faktycznie tworzy krotkę, przynajmniej w sytuacjach, gdy nie mogą być interpretowane w inny sposób):

```
>>> T = 'mielonka', 3.0, [11, 22, 33]  
>>> T[1]  
3.0  
>>> T[2][1]  
22  
>>> T.append(4)  
AttributeError: 'tuple' object has no attribute 'append'
```

Do czego służą krotki

Po co nam zatem typ podobny do listy, który obsługuje mniejszą liczbę operacji? Szczerze mówiąc, w praktyce krotki nie są używane tak często jak listy, ale ich najważniejszą zaletą jest niemutowalność. Kiedy w programie przekazujemy zbiór obiektów w postaci listy, takie obiekty

mogą się w dowolnym momencie zmienić. W przypadku użycia krotki zmienić się nie mogą. Krotki zapewniają pewne ograniczenia w zakresie integralności, które są bardzo wygodne w programach większych niż te, które będziemy tutaj pisać. O krotkach powiemy jeszcze w dalszej części książki (w tym o opartych na krotkach rozszerzeniach, noszących nazwę *nazwane krotki*). Zanim to jednak nastąpi, omówimy jeszcze nasz ostatni typ podstawowy, czyli pliki.

Pliki

Obiekty plików są w Pythonie głównym interfejsem do plików znajdujących się na komputerze. Można ich używać do odczytywania i zapisywania notatek tekstowych, klipów audio, dokumentów Excela, wiadomości e-mail i wszystkiego innego, co zdarzyło Ci się przechowywać na swoim komputerze. Są typem podstawowym, jednak nieco innym od pozostałych. Nie istnieje żadna jednoznacznie określona składnia do ich tworzenia. Zamiast tego do utworzenia obiektu wywołuje się wbudowaną funkcję `open`, przekazując do niej nazwę pliku oraz ciąg znaków określający tryb przetwarzania.

Na przykład, aby utworzyć plik wyjściowy, powinieneś wywołać funkcję `open`, przekazując do niej nazwę pliku wraz z łańcuchem znaków '`w`' określającym tryb przetwarzania umożliwiający zapis danych:

```
>>> f = open('data.txt', 'w')          # Utworzenie nowego pliku w trybie do zapisu  
>>> f.write('Witaj,\n')            # Zapisanie ciągu znaków w pliku  
6  
>>> f.write('Brian\n')           # w Pythonie 3.x funkcja write zwraca liczbę zapisanych bajtów  
6  
>>> f.close()                  # Zapisanie bufora wyjściowego na dysku i zamknięcie pliku
```

Powyższy kod tworzy plik w katalogu bieżącym i zapisuje do niego tekst (jeżeli chcesz uzyskać dostęp do pliku znajdującego się w innej lokalizacji, nazwa pliku może być pełną ścieżką do odpowiedniego katalogu). Aby z powrotem załadować zawartość pliku, należy ponownie go otworzyć, tym razem w trybie do odczytu '`r`' (jest to wartość domyślna, jeżeli pominiemy tryb w wywołaniu). Następnie musimy załadować zawartość pliku do obiektu tekstopwego i wyświetlić ją. Zawartość pliku jest zawsze dla skryptu ciągiem znaków, niezależnie od typu danych zapisanych w takim pliku.

```
>>> f = open('data.txt')          # tryb do odczytu ('r') jest domyślnym trybem przetwarzania plików  
>>> text = f.read()             # załadowanie całego pliku do obiektu tekstopwego  
>>> text  
'Witaj,\nBrian\n'  
>>> print(text)                # funkcja print interpretuje znaki sterujące  
Witaj,  
Brian
```

```
>>> text.split()                                # zawartość pliku jest zawsze łańcuchem
znaków
['Witaj,', 'Brian']
```

Inne metody plików obsługują dodatkowe operacje, których nie mamy teraz czasu omawiać. Obiekty plików udostępniają na przykład większą liczbę sposobów odczytywania i zapisywania (metoda `read` przyjmuje opcjonalny rozmiar w bajtach, a `readline` odczytuje dane z pliku po jednym wierszu naraz), a także inne narzędzia (metoda `seek` przechodzi do nowej pozycji w pliku). Jak jednak zobaczyłeś później, najlepszym sposobem na odczytanie zawartości pliku jest *niewczytywanie go* — pliki udostępniają *iterator*, który automatycznie odczytuje wiersz po wierszu w pętlach `for` oraz innych kontekstach:

```
>>> for line in open('data.txt'): print(line)
```

Z pełnym zestawem metod plików spotkamy się w dalszej części książki, ale jeżeli chcesz już teraz przyjrzeć się tym metodom, możesz wywołać funkcję `dir` dla dowolnego otwartego pliku, a następnie funkcję `help` dla dowolnej ze zwróconych metod:

```
>>> dir(f)
[ ...pominięto wiele nazw...
'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno',
'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read',
'readable', 'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate',
'writable', 'write', 'writelines']
>>> help(f.seek)
...przekonaj się sam!...
```

Pliki binarne

W poprzednim podrozdziale omówiliśmy podstawowe zagadnienia związane z operacjami na plikach, które wystarczają do wykonania wielu zadań. Z technicznego punktu widzenia wykorzystują one jednak domyślne kodowanie Unicode w Pythonie 3.x lub 8-bitowe w Pythonie 2.x. Pliki tekstowe w wersji 3.x są zawsze zamieniane na Unicode w wersji 3.x i zapisywane wprost w wersji 2.x. Nie ma to znaczenia dla wcześniej używanych prostych danych ASCII, które niezmiennie odwzorowują bajty z pliku na znaki i odwrotnie, jednak w przypadku bogatszych typów danych interfejsy plików mogą się różnić w zależności od zawartości i używanej wersji Pythona.

Jak już wspominaliśmy wcześniej, Python 3.x wprowadza wyraźne rozróżnienie między tekstem a danymi binarnymi w plikach: w *plikach tekstowych* przechowywane są normalne łańcuchy znaków `str`, automatycznie kodowane i dekodowane z i na Unicode podczas zapisywania i odczytywania danych, podczas gdy zawartość *plików binarnych* jest reprezentowana jako ciąg bajtowy `bytes`. W Pythonie 2.x znajdziemy podobne rozróżnienie, ale nie jest ono narzucone tak sztywno, a dostępne narzędzia również są nieco inne.

Na przykład *pliki binarne* są przydatne do przetwarzania multimedialnych, uzyskiwania dostępu do danych utworzonych przez programy w języku C i tak dalej. Na przykład moduł `struct` Pythona pozwala zarówno na tworzenie, jak i rozpakowywanie *danych binarnych* (czyli surowych wartości bajtów niebędących obiektami Pythona), które będą zapisywane w plikach w trybie binarnym. Szczegółowo przestudujemy tę technikę w dalszej części książki, ale koncepcja jest prosta — w przykładzie pokazanym poniżej używamy Pythona 3.x i tworzymy plik binarny (w wersji 2.x pliki binarne działają tak samo, ale przedrostek `b` literału nie jest wymagany i nie będzie wyświetlany):

```
>>> import struct
```

```

>>> packed = struct.pack('>i4sh', 7, b'spam', 8)      # Tworzymy spakowane
dane binarne

>>> packed                                         # 10 bajtów, to nie
obiekt ani tekst

b'\x00\x00\x00\x07spam\x00\x08'

>>>

>>> file = open('data.bin', 'wb')                  # Otwieramy binarny plik
wyjściowy

>>> file.write(packed)                            # Zapisujemy spakowane
dane binarne

10

>>> file.close()

```

Odczytywanie danych binarnych jest zasadniczo symetryczne; nie wszystkie programy muszą pracować z plikami na tak niskim poziomie, ale mechanizm plików binarnych powoduje, że w Pythonie jest to łatwe:

```

>>> data = open('data.bin', 'rb').read()            # Otwieramy i
odczytujemy plik binarny

>>> data                                         # 10 bajtów, bez
modyfikacji

b'\x00\x00\x00\x07spam\x00\x08'

>>> data[4:8]                                     # Wycinamy bajty ze
środku

b'spam'

>>> list(data)                                    # Sekwencja bajtów 8-
bitowych

[0, 0, 0, 7, 115, 112, 97, 109, 0, 8]

>>> struct.unpack('>i4sh', data)                  # Rozpakowujemy ponownie
do postaci obiektu

(7, b'spam', 8)

```

Pliki tekstowe Unicode

Pliki tekstowe są wykorzystywane do przechowywania i przetwarzania wszelkiego rodzaju danych tekstowych, od notatek, poprzez treści wiadomości e-mail, aż po dokumenty JSON i XML. Jednak w dzisiejszym, szeroko połączonym świecie nie możemy tak naprawdę rozmawiać o danych tekstowych bez zadania pytania o ich format — musimy znać typ kodowania tekstu w Unicode, jeżeli różni się on od domyślnego ustawienia naszego komputera lub nie możemy polegać na ustawieniach domyślnych ze względu na konieczność przenoszenia danych na inne systemy i platformy.

Na szczęście jest to łatwiejsze, niż się na pierwszy rzut oka może wydawać. Aby uzyskać dostęp do plików zawierających *tekst Unicode* inny niż ASCII (np. taki jak pokazywaliśmy wcześniej w tym rozdziale), po prostu przekazujemy nazwę kodowania (jeżeli tekst w pliku nie jest zgodny z domyślnym kodowaniem dla naszej platformy). W takim trybie Python automatycznie *koduje*

pliki tekstowe podczas zapisywania i *dekoduje* podczas odczytywania zgodnie z podaną nazwą schematu kodowania. W *Pythonie 3.x* wygląda to tak:

```
>>> S = 'sp\xc4m'                                # Tekst Unicode (spoza
       zestawu ASCII)

>>> S
'spÄm'

>>> S[2]                                         # Sekwencja znaków
'Ä'

>>> file = open('unidata.txt', 'w', encoding='utf-8') # Zapisywanie tekstu w
       formacie UTF-8

>>> file.write(S)                                 # 4 znaki zostały
       zapisane

4

>>> file.close()

>>> text = open('unidata.txt', encoding='utf-8').read() # Odczytywanie tekstu
       i dekodowanie UTF-8

>>> text
'spÄm'

>>> len(text)                                     # 4 znaki (kody
       znaków)

4
```

Takie automatyczne kodowanie i dekodowanie jest tym, czego zwykle oczekujesz. Ponieważ pliki obsługują odpowiednie kodowanie i dekodowanie podczas zapisywania i odczytywania, możesz przetwarzać tekst w pamięci jako zwykłe ciągi znaków bez martwienia się o to, że był zakodowany w Unicode. W razie potrzeby możesz jednak zobaczyć, co naprawdę jest zapisane w pliku. Aby to zrobić, powinieneś przejść do trybu binarnego:

```
>>> raw = open('unidata.txt', 'rb').read()      # Odczytujemy surowe wartości
       bajtów

>>> raw
b'sp\xc3\x84m'

>>> len(raw)                                      # Naprawdę w formacie UTF-8
       zostało zapisanych 5 bajtów

5
```

Jeżeli otrzymujesz dane Unicode ze źródła innego niż plik, na przykład z wiadomości e-mail czy poprzez połaczenie sieciowe, możesz je kodować i dekodować ręcznie, na przykład:

```
>>> text.encode('utf-8')                         # Ręczne kodowanie na ciąg
       bajtów UTF-8

b'sp\xc3\x84m'

>>> raw.decode('utf-8')                          # Ręczne dekodowanie na ciąg
       znaków str
```

```
'spÄm'
```

Warto również sprawdzić, w jaki sposób pliki tekstowe automatycznie kodowałyby ten sam ciąg znaków w różnych schematach kodowania — w plikach na dysku zapisywane będą różne ciągi bajtów, które jednak po podaniu poprawnej nazwy schematu kodowania będą dekodowane do postaci tego samego, początkowego ciągu znaków:

```
>>> text.encode('latin-1')                                # Ciągi bajtów są różne w
różnych

# schematach kodowania
b'sp\xc4m'

>>> text.encode('utf-16')
b'\xff\xfe\x00p\x00\xc4\x00m\x00'

>>> len(text.encode('latin-1')), len(text.encode('utf-16'))
(4, 10)

>>> b'\xff\xfe\x00p\x00\xc4\x00m\x00'.decode('utf-16') # Ale po zdekodowaniu
to wciąż ten sam

# ciąg znaków
'spÄm'
```

To wszystko działa mniej więcej tak samo w Pythonie 2.x, ale ciągi Unicode są kodowane i wyświetlane z wiodącym znakiem u, ciągi bajtów nie wymagają znaku wiodącego lub są wyświetlane z wiodącym b, a pliki tekstowe Unicode muszą być otwierane za pomocą funkcji `codecs.open`, która akceptuje nazwę schematu kodowania (tak jak funkcja `open` w wersji 3.x) i używa specjalnego ciągu `unicode` do reprezentowania zawartości w pamięci. Tryb pliku binarnego może wydawać się opcjonalny w wersji 2.x, ponieważ normalne pliki to tylko bajty, ale powinieneś unikać modyfikowania znaków końca linii, jeżeli są obecne (więcej szczegółowych informacji na ten temat znajdziesz w dalszej części książki):

```
>>> import codecs

>>> codecs.open('unidata.txt', encoding='utf8').read()      # 2.x: odczytywanie
i dekodowanie tekstu
u'sp\xc4m'

>>> open('unidata.txt', 'rb').read()                          # 2.x: odczytywanie
surowych bajtów
'sp\xc3\x84m'

>>> open('unidata.txt').read()                               # 2x: surowe,
niedekodowane bajty
'sp\xc3\x84m'
```

Jeżeli zajmujesz się tylko tekstami w standardzie ASCII, to nie będziesz musiał się przejmować opisanymi wyżej różnicami w sposobach obsługi różnych zestawów znaków. Warto jednak zauważyc, że jeżeli masz do czynienia z danymi binarnymi (co obejmuje większość plików multimedialnych) lub tekstami wykorzystującymi międzynarodowe zestawy znaków (co obecnie jest standardem dla większość treści dostępnych w internecie), to mechanizmy obsługi ciągów i plików zaimplementowane w Pythonie są jego ogromnym atutem. Python obsługuje również nazwy plików (a nie tylko zawartość) zawierające znaki spoza standardowego zestawu ASCII i

ułatwia automatyzację wielu zadań, a jego narzędzia, takie jak *walker* i *lister*, dają programistom dużą elastyczność. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 37.

Inne narzędzia podobne do plików

Funkcja `open` jest koniem pociągowym dla większości zadań związanych z przetwarzaniem plików w Pythonie. Dla bardziej zaawansowanych zadań Python udostępnia dodatkowe narzędzia podobne do plików: potoki, kolejki FIFO, gniazda, pliki zabezpieczane przy użyciu klucza/hasła, obiekty trwałe, pliki deskryptorów czy interfejsy do relacyjnych i zorientowanych obiektowo baz danych i inne. Pliki deskryptorów obsługują na przykład blokowanie pliku i inne narzędzia niskiego poziomu, natomiast gniazda udostępniają interfejs do zadań sieciowych i komunikacji międzyprocesowej. Co prawda wielu z tych zagadnień nie będziemy omawiać w tej książce, ale z pewnością przydadzą Ci się one, gdy zaczniesz programować w Pythonie na poważnie.

Inne typy podstawowe

Poza omówionymi dotychczas typami podstawowymi istnieją inne, które mogą, ale nie muszą się zaliczać do tej kategorii, w zależności od tego, jak szeroko ją zdefiniujemy. Nowym dodatkiem do języka są na przykład *zbiory*, które nie są ani odwzorowaniem, ani sekwencjami. Zbiory to raczej nieuporządkowane kolekcje unikalnych i niemutowalnych obiektów. Tworzymy je, wywołując wbudowaną funkcję `set` lub za pomocą nowych literałów i wyrażeń zbiorów w Pythonie 2.7 i 3.x. Na zbiorach można wykonywać standardowe operacje matematyczne. Nowa składnia `{...}` dla literałów zbiorów nie jest przypadkowa, ponieważ zbiory przypominają klucze słownika bez wartości.

```
>>> X = set('mielonka')                                # Utworzenie zbioru z sekwencji w
wersjach 2.x oraz 3.x

>>> Y = {'s', 'z', 'y', 'n', 'k', 'a'}    # Utworzenie zbioru za pomocą nowego
literatu w wersjach 3.x i 2.x

>>> X, Y                                         # Krotka z dwóch zbiorów bez nawiasów
({'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n'}, {'a', 'k', 'n', 's', 'y', 'z'})

>>> X & Y                                     # Część wspólna zbiorów
{'a', 'k', 'n'}

>>> X | Y                                     # Suma zbiorów
{'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n', 's', 'y', 'z'}

>>> X - Y                                     # Różnica
{'i', 'm', 'e', 'l', 'o'}

>>> X > Y                                     # Nadzbiór
False

>>> {x ** 2 for x in [1, 2, 3, 4]}      # Zbiory składane w wersji 3.x i 2.7
{16, 1, 4, 9}
```

Nawet mniej uzdolnieni matematycznie programiści często korzystają ze zbiorów, które są przydatne do wykonywania wielu typowych zadań, takich jak filtrowanie duplikatów, izolowanie

różnic i wykonywanie niezależnych od kolejności elementów testów równości bez konieczności sortowania na listach, ciągach i wszystkich innych obiektach iterowalnych:

```
>>> list(set([1, 2, 1, 3, 1]))          # Filtrowanie duplikatów (elementy nie
muszą być uporządkowane)
[1, 2, 3]
>>> set('mielonka') - set('szynka')    # Wyszukiwanie różnic w kolekcjach
{'o', 'i', 'm', 'l', 'e'}
>>> set('mielonka') == set('alkmoeni') # Testy równości niezależne od
kolejności elementów
True
```

Zbiory obsługują również testy przynależności, chociaż wszystkie inne rodzaje kolekcji w Pythonie również mają takie możliwości:

```
>>> 'n' in set('mielonka'), 'n' in 'mielonka', 'szynka' in ['jajka',
'mielonka', 'szynka']
(True, True, True)
```

Ponadto w Pythonie niedawno pojawiło się kilka nowych typów liczbowych: liczby *dziesiętne*, które są liczbami zmiennoprzecinkowymi o stałej precyzyji, oraz liczby *ułamkowe*, które są liczbami wymiernymi z licznikiem i mianownikiem. Oba typy można wykorzystać do obejścia ograniczeń i niedokładności będących nieodłączną częścią arytmetyki liczb zmiennoprzecinkowych.

```
>>> 1 / 3                                # Liczby zmiennoprzecinkowe (w
Pythonie 2.x należy dodać .0)
0.3333333333333333
>>> (2/3) + (1/2)
1.1666666666666665
>>> import decimal                      # Liczby dziesiętne – stała precyzja
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal('4.141')
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33')
>>> from fractions import Fraction      # Ułamki – licznik i mianownik
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)
```

Python udostępnia także wartości typu *Boolean* (ze zdefiniowanymi obiektami `True` i `False`, które są tak naprawdę liczbami całkowitymi o wartościach odpowiednio 1 i 0 z własną logiką wyświetlania) i od dawna obsługuje specjalny obiekt zastępczy o nazwie `None`, wykorzystywany najczęściej do inicjowania zmiennych i obiektów:

```
>>> 1 > 2, 1 < 2                                # Wartości typu Boolean
(False, True)
>>> bool('mielonka')                            # Wartość logiczna obiektu
True
>>> X = None                                    # Obiekt None
>>> print(X)
None
>>> L = [None] * 100                             # Zainicjowanie listy stu obiektów
None
>>> L
[None, None, None,
None, ...lista stu obiektów None...]
```

Jak zepsuć elastyczność kodu

Więcej o obiektach Pythona powiemy w dalszej części książki, jednak jeden z nich zasługuje na kilka słów już teraz. Obiekt *typu*, zwracany przez funkcję wbudowaną `type`, jest obiektem zwracającym typ innego obiektu. Jego działanie różni się nieco w Pythonie 3.x, gdzie typy zostały całkowicie scalone z klasami (będziemy o tym mówić w szóstej części książki w kontekście klas w nowym stylu). Zakładając, że `L` nadal jest listą z poprzedniego podrozdziału:

```
# W Pythonie 2.x:
>>> type(L)                                     # Typy: L jest obiektem typu lista
<type 'list'>
>>> type(type(L))                               # Typy też są obiektami
<type 'type'>
# W Pythonie 3.x:
>>> type(L)                                     # 3.x: typy są klasami i odwrotnie
<class 'list'>
>>> type(type(L))                               # Więcej na temat typów klas
znajdziesz w rozdziale 32.
<class 'type'>
```

Poza umożliwieniem interaktywnego badania innych obiektów, obiekt `type` pozwala również na sprawdzanie typów przetwarzanych obiektów z poziomu kodu programu. W skryptach Pythona można to zrobić na przynajmniej trzy sposoby.

```
>>> if type(L) == type([]):                      # Sprawdzanie typu obiektu (skoro
już musimy...)
    print('tak')
```

```

tak

>>> if type(L) == list:                      # Użycie nazwy typu
    print('tak')

tak

>>> if isinstance(L, list):                  # Sprawdzanie zorientowane obiektywne
    print('tak')

tak

```

Pokazałem wszystkie te sposoby sprawdzania typów, jednak moim obowiązkiem jest dodać, że korzystanie z nich jest prawie zawsze złym pomysłem w programie napisanym w Pythonie (i często jest też znakiem rozpoznawczym byłego programisty języka C, który zaczyna przygodę z Pythonem). Przyczyna takiego stanu rzeczy stanie się całkowicie zrozumiała dopiero później, kiedy zacznijemy pisać większe jednostki kodu, takie jak funkcje, jednak jest to koncepcja kluczowa dla Pythona (być może wręcz najważniejsza ze wszystkich). Sprawdzając określone typy w kodzie, bardzo efektywnie niszczymy jego elastyczność, ograniczając go do tylko jednego typu danych. Bez takiego sprawdzania kod może działać na szerokiej gamie typów danych.

Jest to związane ze wspomnianą wcześniej koncepcją *polimorfizmu* i ma swoje źródło w braku deklaracji typu w Pythonie. W Pythonie, czego nauczymy się już niedługo, tworzymy kod pod kątem *interfejsów obiektów* (obsługiwanych operacji), a nie typów. Inaczej mówiąc, oznacza to, że dbamy o to, co *robi* obiekt, a nie czym *jest*. Nieprzejmowanie się konkretnymi typami sprawia, że kod automatycznie można zastosować do wielu typów danych. Każdy obiekt ze zgodnym interfejsem będzie działał bez względu na typ. Choć sprawdzanie typów jest obsługiwane — a nawet w niektórych przypadkach wymagane — to nie jest to „pythonowy” sposób myślenia. Jak się okaże, sam polimorfizm jest jedną z kluczowych koncepcji leżących u podstaw używania Pythona.

Klasy definiowane przez użytkownika

W dalszej części książki będziemy bardziej szczegółowo omawiali *programowanie zorientowane obiektywne* w Pythonie — opcjonalną, lecz mającą duże możliwości właściwość tego języka pozwalającą na skrócenie czasu programowania dzięki dostosowaniu obiektów do własnych potrzeb. Z abstrakcyjnego punktu widzenia klasy definiują nowe typy obiektów, które rozszerzają zbiór typów podstawowych, dlatego zasługują na kilka słów w tym miejscu. Powiedzmy na przykład, że potrzebny jest nam typ obiektu będący modelem pracownika. Choć taki właśnie typ nie istnieje w Pythonie, poniższa zdefiniowana przez użytkownika klasa powinna się przydać.

```

>>> class Worker:

    def __init__(self, name, pay):      # Inicjalizacja przy utworzeniu
        self.name = name                # self jest nowym obiektem; name
        to nazwisko, a pay – płaca

        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]     # Podział łańcucha na pustych
        znakach

    def giveRaise(self, percent):

```

```
self.pay *= (1.0 + percent)      # Uaktualnienie płacy w miejscu
```

Powysza klasa definiuje nowy rodzaj obiektu, który będzie miał atrybuty `name` (nazwisko) i `pay` (płaca) — czasami nazywane *informacjami o stanie* (ang. *state information*) — a także dwa rodzaje zachowania zakodowane w postaci funkcji (normalnie nazywanych *metodami*). Wywołanie klasy w sposób podobny do funkcji generuje obiekty naszego nowego typu, a metody klasy automatycznie otrzymują instancję obiektu przetwarzanego przez dane wywołanie metody (w argumencie `self`).

```
>>> bob = Worker('Robert Zielony', 50000)      # Utworzenie dwóch instancji
objektu

>>> anna = Worker('Anna Czerwona', 60000)      # Każdy obiekt ma atrybuty w
postaci imienia i nazwiska

# osoby oraz płacy

>>> bob.lastName()                                # Wywołanie metody – self to bob
'Zielony'

>>> anna.lastName()                                # Teraz self to anna
'Czerwona'

>>> anna.giveRaise(.10)                            # Uaktualnienie płacy Anny

>>> anna.pay

66000.0
```

Domniemany obiekt `self` jest przyczyną nazwania tego modelu *zorientowanym obiektem* — w funkcjach klasy zawsze istnieje domniemany podmiot. W pewnym sensie typ oparty na klasie po prostu tworzy coś na bazie typów podstawowych. Zdefiniowany wyżej obiekt `Worker` jest na przykład zbiorem składającym się z łańcucha znaków oraz liczby (odpowiednio `name` i `pay`) wraz z funkcjami odpowiedzialnymi za przetwarzanie tych wbudowanych obiektów.

Bardziej rozbudowane wyjaśnienie kwestii klas mówi, że to ich mechanizm dziedziczenia obsługuje hierarchię oprogramowania, która pozwala na dostosowywanie programów do własnych potrzeb poprzez *rozszerzanie* ich. Programy rozszerza się poprzez pisanie nowych klas, a nie modyfikowanie tego, co już działa. Warto również wiedzieć, że klasy są w Pythonie opcjonalne i w wielu przypadkach proste obiekty wbudowane, takie jak listy i słowniki, są często lepszymi narzędziami od klas zakodowanych przez użytkowników. Wszystko to jednak wykracza poza zakres wstępnego omówienia typów, dlatego powinieneś uznać to tylko za zapowiedź tego, co znajdziesz w dalszej części książki. Ponieważ klasy opierają się na innych narzędziach języka Python, są one jednym z głównych zagadnień omawianych w tej książce.

I wszystko inne

Jak wspomniano wcześniej, wszystko, co przetwarzamy w skryptach Pythona, jest rodzajem obiektu, przez co nasze omówienie z pewnością nie jest kompletne. Jednak mimo, że wszystko w Pythonie jest obiektem, jedynie obiekty przedstawione wyżej stanowią zbiór obiektów podstawowych tego języka. Inne typy w Pythonie to obiekty związane z wykonywaniem programu (takie jak funkcje, moduły, klasy i skompilowany kod), które będziemy badać później, lub są implementowane przez funkcje importowanych modułów, a nie składnię języka. Ta druga kategoria pełni zwykle role specyficzne dla określonych zastosowań — wzorców tekstowych, interfejsów do baz danych czy połączeń sieciowych.

Należy również pamiętać, że obiekty omówione wyżej są co prawda *obiektami*, ale niekoniecznie są *zorientowanymi obiektemi*. Zorientowanie obiektowe zazwyczaj wymaga w

Pythonie dziedziczenia i instrukcji `class`, z którą spotkamy się w dalszej części książki. Obiekty podstawowe Pythona są siłą napędową prawie każdego skryptu napisanego w tym języku, z jakim można się spotkać, i zazwyczaj są też podstawą dla bardziej rozbudowanych typów, które nie mieszą się w tej kategorii.

Podsumowanie rozdziału

I to by było na tyle, jeżeli chodzi o naszą krótką wycieczkę po typach danych. W niniejszym rozdziale zamieściłem zwięzłe wprowadzenie do podstawowych typów obiektów w Pythonie wraz z omówieniem rodzajów operacji, jakie można do nich zastosować. Przyjrzaliśmy się uniwersalnym operacjom, które działają na wielu typach obiektów (na przykład operacjom na sekwencjach, takim jak indeksowanie czy wycinki), a także operacjom specyficznych dla określonego typu, dostępnym w postaci wywołania metody (na przykład dzieleniełańcuchów znaków czy dodawanie elementów do listy). Zdefiniowaliśmy również pewne kluczowe pojęcia, takie jak niezmienność, sekwencje i polimorfizm.

Po drodze widzieliśmy również, że podstawowe typy obiektów Pythona są bardziej elastyczne i mają większe możliwości od tego, co dostępne jest w językach niższego poziomu, takich jak C. Listy i słowniki w Pythonie usuwają na przykład większość pracy koniecznej do obsługiwanego kolekcji oraz wyszukiwania w językach niższego poziomu. Listy to uporządkowane kolekcje innych obiektów, natomiast słowniki to kolekcje innych obiektów indeksowane po kluczu, a nie pozycji. Zarówno słowniki, jak i listy mogą być zagnieżdżane, mogą się rozszerzać i kurczyć na życzenie i mogą zawierać obiekty dowolnego typu. Co więcej, po porzuceniu ich automatycznie odzyskiwane jest miejsce zajmowane przez nie w pamięci. Zauważaliśmy również, że ciągi znaków i pliki współpracują ze sobą, obsługując bogatą różnorodność danych binarnych i tekstowych.

Aby zachować zwięzłość i rozsądne rozmiary tego rozdziału, w opisach poszczególnych zagadnień celowo pominąłem wiele szczegółów, dlatego nie powinieneś oczekwać, że całość tego rozdziału będzie miała dla Ciebie sens. W kolejnych rozdziałach zaczniemy kopać coraz głębiej i głębiej, odkrywając szczegółowe podstawowe typy obiektów Pythona, które tutaj pominęliśmy w celu lepszego zrozumienia całości. Zaczniemy już w następnym rozdziale od pogłębionego omówienia liczb w Pythonie. Najpierw jednak — kolejny quiz.

Sprawdź swoją wiedzę — quiz

Koncepcje przedstawione w niniejszym rozdziale omówimy bardziej szczegółowo w kolejnych rozdziałach, dlatego teraz czas na szerszą perspektywę.

1. Podaj cztery podstawowe typy danych Pythona.
2. Dlaczego typy te nazywane są podstawowymi?
3. Co oznacza pojęcie „niemutowalny” i które trzy z podstawowych typów danych Pythona są uznawane za niemutowalne?
4. Czym jest sekwencja i które trzy typy danych należą do tej kategorii?
5. Czym jest odwzorowanie i który typ podstawowy zalicza się do tej kategorii?
6. Czym jest polimorfizm i dlaczego powinno to być dla nas ważne?

Sprawdź swoją wiedzę – odpowiedzi

1. Liczby, łańcuchy znaków, listy, słowniki, krotki, pliki i zbiory są uważane za podstawowe typy danych w Pythonie. Czasami w ten sam sposób klasyfikowane są również typy, obiekty `None` i wartości typu Boolean. Istnieje kilka typów liczbowych (liczby całkowite, zmiennoprzecinkowe, zespolone, ułamkowe i dziesiętne), a także różne typy łańcuchów znaków (zwykle łańcuchy znaków, łańcuchy znaków Unicode z Pythona 2.x, łańcuchy tekstowe oraz bajtowe z Pythona 3.x).
2. Wymienione wyżej typy nazywane są podstawowymi, ponieważ są częścią samego języka i są zawsze dostępne. Aby utworzyć inne typy obiektów, konieczne jest wywołanie funkcji z importowanych modułów. Większość typów podstawowych ma określzoną składnię służącą do ich generowania — na przykład '`mielonka`' to wyrażenie tworzące łańcuch znaków i ustalające zbiór operacji, które mogą być do niego zastosowane. Z tego powodu typy podstawowe są ściśle połączone ze składnią Pythona. W przeciwnieństwie do tego musisz wywołać wbudowaną funkcję `open`, aby utworzyć obiekt pliku (nawet jeżeli zwykle jest on również uważany za typ podstawowy).
3. Obiekt niemutowalny to taki obiekt, który nie może być zmodyfikowany po utworzeniu. W Pythonie do tej kategorii zaliczamy liczby, łańcuchy znaków i krotki. Choć obiektu niemutowalnego nie można modyfikować w miejscu, zawsze można utworzyć nowy obiekt, wykonując odpowiednie wyrażenie. Typ `bytearray`, dostępny w najnowszych wersjach Pythona, oferuje co prawda mutowalność tekstu, ale ciągi tego typu nie są normalnymi ciągami znaków i odnoszą się bezpośrednio do tekstu tylko wtedy, gdy składa się z prostych 8-bitowych znaków (np. ASCII).
4. Sekwencja to uporządkowana pod względem pozycji kolekcja obiektów. W Pythonie sekwencjami są łańcuchy znaków, listy oraz krotki. Dzielą one wspólne operacje na sekwencjach, takie jak indeksowanie, konkatenacja czy wycinki, jednak każdy ma również specyficzne dla danego typu metody. Powiązany termin „iterowalny” oznacza sekwencję fizyczną lub wirtualną, która generuje swoje elementy na żądanie.
5. Pojęcie „odwzorowanie” oznacza obiekt, który dokonuje mapowania kluczy na powiązane z nimi wartości. Jedynym takim typem w zbiorze podstawowych typów obiektów Pythona jest słownik. Odwzorowania nie zachowują żadnej stałej kolejności elementów od lewej do prawej strony. Obsługują dostęp do danych według klucza oraz metody specyficzne dla danego typu.
6. Polimorfizm oznacza, że znaczenie operacji (takiej jak `+`) uzależnione jest od obiektów, na których się ona odbywa. Jest to kluczowa koncepcja stanowiąca podstawkę Pythona (być może nawet tę najważniejszą) — nieograniczanie kodu do określonych typów sprawia, że kod ten można automatycznie zastosować do wielu typów.

[1] W niniejszej książce pojęcie *literal* oznacza po prostu wyrażenie, którego składnia generuje obiekt — czasami nazywane również *stałą*. Warto zauważyć, że „stała” nie oznacza wcale obiektów czy zmiennych, których nie można modyfikować (czyli pojęcie to nie ma związku z elementami typu `const` z języka C++ czy niemutowalnymi [ang. *immutable*] obiektami Pythona, które zostaną omówione w dalszej części rozdziału).

[2] Taka struktura macierzy dobrze sprawdza się podczas wykonywania niewielkich zadań, jednak w przypadku poważniejszego przetwarzania danych liczbowych lepiej będzie skorzystać z jednego z numerycznych rozszerzeń Pythona, takich jak pakiety *NumPy* czy *SciPy* na licencji open source. Tego typu narzędzia są w stanie przechowywać i przetwarzać duże macierze o wiele bardziej wydajnie niż zagnieżdżona struktura listy. Często możesz spotkać się ze

stwierdzeniem, że pakiet NumPy zmienia Pythona w darmowy i bardziej zaawansowany odpowiednik systemu MatLab; warto przypomnieć, że firmy i organizacje takie jak NASA, Los Alamos, JPL i wiele innych wykorzystują takie pakiety w swojej pracy naukowej i finansowej. Więcej szczegółowych informacji na ten temat możesz znaleźć w internecie.

[3] w oryg. *something completely different*; autor nawiązuje tutaj do nakręconego w roku 1971 filmu *And Now for Something Completely Different*, który oparty został na serialu komediowym „Monty Python’s Flying Circus” — przyp. tłum.

[4] Dwie uwagi: należy pamiętać, że utworzony przed chwilą rekord `rec` mógłby być prawdziwym rekordem bazy danych, gdybyśmy użyli mechanizmu *utrwalania obiektów* (ang. *object persistence*) Pythona — łatwego sposobu przechowywania obiektów Pythona w plikach lub bazach danych, który automatycznie dokonuje serializacji i deserializacji obiektów. Nie będziemy teraz omawiać szczegółów tego rozwiązania; więcej szczegółowych informacji na temat utrwalania obiektów znajdziesz w podrozdziałach dotyczących modułów `pickle` oraz `shelve`, które znajdziesz w rozdziale 9., rozdziale 28., rozdziale 31. i rozdziale 37., gdzie będziemy je omawiać w kontekście odpowiednio: zastosowania do plików tekstowych, programowania obiektowego, klas i zmian, jakie w nich zostały zaimplementowane w wersji 3.x.

Po drugie, jeżeli znasz format JSON (ang. *JavaScript Object Notation*) — popularny format wymiany danych używany do baz danych i transferów sieciowych — ten przykład może Ci się wydawać bardzo podobny, chociaż sposób, w jaki Python obsługuje zmienne, wyrażenia i zmiany, może sprawić, że używane struktury danych będą bardziej ogólne. Moduł biblioteki `json` Pythona obsługuje tworzenie i analizowanie kodu JSON, ale zamiana takiego kodu na obiekty Pythona jest często trywialnie prosta. Zwróć uwagę na przykład z formatem JSON w rozdziale 9., który używa tego samego rekordu. W przypadku większych ilości danych możesz użyć na przykład bazy danych *MongoDB*, która przechowuje dane w postaci zakodowanych binarnie serializowanych dokumentów podobnych do formatu JSON. Do korzystania z tej bazy z poziomu Pythona możesz użyć interfejsu *PyMongo*.

Rozdział 5. Typy liczbowe

Niniejszy rozdział rozpoczyna nasze pogłębione omówienie języka Python. W Pythonie dane przybierają formę *obiektów* — albo wbudowanych obiektów udostępnianych przez język, albo obiektów tworzonych za pomocą narzędzi Pythona i innych języków, takich jak C. Obiekty są tak naprawdę podstawą każdego programu, jaki pisze się w Pythonie. Ponieważ są podstawowym budulcem tego języka, będą również stanowiły nasz pierwszy obiekt zainteresowania.

W poprzednim rozdziale krótko przedstawiliśmy podstawowe typy obiektów Pythona. Choć zostały w nim wprowadzone najważniejsze pojęcia, nie zagłębialiśmy się w szczególności. Teraz zaczniemy przyglądać się bardziej dokładnie różnym koncepcjom związanym ze strukturami danych, tak by wypełnić pominięte wcześniej detale. Zaczniemy zatem od naszej pierwszej kategorii typów danych Pythona — typów liczbowych i operacji numerycznych.

Podstawy typów liczbowych Pythona

Większość typów liczbowych Pythona jest dość typowa i dla osób używających kiedyś jakiegokolwiek języka programowania powinny one wyglądać znajomo. Mogą być wykorzystywane do przechowywania stanu konta, odległości z Ziemi do Marsa, liczby odwiedzających naszą witrynę internetową i każdej innej wartości liczbowej.

W Pythonie liczby nie są tak naprawdę jednym typem obiektu; są raczej kategorią podobnych typów. Python obsługuje zwykle typy liczbowe (liczby całkowite oraz zmiennoprzecinkowe), a także literaly służące do tworzenia liczb i wyrażenia je przetwarzające. Dodatkowo Python udostępnia bardziej zaawansowaną obsługę programowania numerycznego, a także obiekty przeznaczone do bardziej zaawansowanych działań. Pełen zasób typów liczbowych Pythona obejmuje:

- obiekty całkowite i zmiennoprzecinkowe;
- obiekty zespolone;
- obiekty dziesiętne o ustalonej precyzji;
- ułamkowe obiekty wymierne;
- zbiory: kolekcje z operacjami numerycznymi;
- wartości typu Boolean: prawda (`true`) i fałsz (`false`);
- wbudowane funkcje i moduły: `round`, `math`, `random` itp.;
- wyrażenia; nieograniczoną precyzję liczb całkowitych; operacje bitowe; formaty szesnastkowe, ósemkowe i binarne;
- rozszerzenia tworzone przez niezależnych programistów: wektory, biblioteki, wizualizacje, tworzenie wykresów itp.

Ponieważ typy wymienione w pierwszym punkcie powyżej listy są chyba najczęściej wykorzystywane w kodzie programów pisanych w języku Python, zaczniemy ten rozdział od omówienia tych podstawowych typów liczbowych, a dopiero potem przejdziemy do eksploracji pozostałych typów na tej liście, mających różne wyspecjalizowane zastosowania. Przyjrzymy się również *zbiorom*, które mają zarówno cechy obiektów liczbowych, jak i kolekcji, ale ogólnie uważa się je bardziej za te pierwsze niż za drugie. Zanim jednak zabierzemy się za pisanie kodu, kilka kolejnych podrozdziałów zajmie nam krótkie omówienie sposobu zapisywania i przetwarzania liczb w naszych skryptach.

Literały liczbowe

Wśród typów podstawowych Python udostępnia *liczby całkowite* (dodatnie i ujemne) oraz *zmiennoprzecinkowe* (z częścią ułamkową). Python pozwala na zapis liczb całkowitych za pomocą literałów szesnastkowych, ósemkowych oraz dwójkowych, oferując dodatkowo typ liczby zespolonej i pozwala, by liczby całkowite miały nieograniczoną *precyzję* (mogą one składać się z takiej liczby cyfr, na jaką pozwoli ilość miejsca w pamięci komputera). Tabela 5.1 pokazuje, jak wyglądają typy liczbowe Pythona, gdy są zapisane w programie jako literały lub wywołania funkcji konstruktora.

Tabela 5.1. Podstawowe literały liczbowe i konstruktory

Literał	Interpretacja
1234, -24, 0, 9999999999999999	Liczby całkowite (ang. <i>integer</i> ; o nieograniczonej wielkości)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Liczby zmiennoprzecinkowe (ang. <i>floating-point number</i>)
0o177, 0x9ff, 0b101010	Literały ósemkowe, szesnastkowe i dwójkowe w wersji 3.x
0177, 0o177, 0x9ff, 0b101010	Literały ósemkowe, szesnastkowe i dwójkowe w wersji 2.x
3+4j, 3.0+4.0j, 3J	Literały liczb zespolonych (ang. <i>complex number</i>)
set('spam'), {1, 2, 3, 4}	Zbiory: tworzenie zbiorów w wersjach 2.x i 3.x
Decimal('1.0'), Fraction(1,3)	Typy rozszerzające: dziesiętne i ułamkowe
bool(X), True, False	Typ Boolean i stałe

Generalnie typy liczbowe Pythona są dość proste pod względem zapisu, choć warto podkreślić kilka kwestii związanych z ich kodowaniem:

Literały liczb całkowitych oraz zmiennoprzecinkowych

Liczby całkowite zapisywane są jak łańcuchy cyfr dziesiętnych. Liczby zmiennoprzecinkowe mają znak dziesiętny (w postaci kropki) lub zawierają opcjonalny wykładnik ze znakiem wprowadzony po literze e lub E, po której znajduje się opcjonalny znak. Jeżeli liczbę zapiszemy ze znakiem dziesiętnym lub wykładnikiem, Python automatycznie robi z niej obiekt liczby zmiennoprzecinkowej i kiedy liczba ta użyta zostanie w wyrażeniu, wykorzysta arytmetykę liczb zmiennoprzecinkowych, a nie całkowitych. Liczby zmiennoprzecinkowe w standardowych dystrybucjach Python zaimplementowane są jako typ *double* z języka C, dlatego mają taką precyzję, jaką kompilator języka C wykorzystany do zbudowania interpretera Pythona przydzieli liczbom tego typu.

Liczby całkowite w Pythonie 2.x — zwykłe i długie

W Pythonie 2.x dostępne są dwa typy liczb całkowitych — zwykłe (zazwyczaj 32-bitowe) oraz długie (o nieograniczonej precyzji). Liczba całkowita może kończyć się znakami l lub L, przez co wymuszane jest potraktowanie jej jako długiej liczby całkowitej. Ze względu na to, że liczby całkowite są automatycznie konwertowane na długie liczby całkowite, kiedy ich wartość przekroczy alokowaną dla nich liczbę bitów (np. 32 bity), nie trzeba wpisywać litery L samodzielnie — w razie potrzeby Python automatycznie dokona konwersji na długą liczbę całkowitą.

Liczby całkowite w Pythonie 3.x — jeden typ

W wersji 3.x dwa typy liczb całkowitych — zwykłe oraz długie — zostały połączone. Istnieje tylko jeden typ liczby całkowitej, który automatycznie obsługuje nieograniczoną precyzję odrębnego typu długiej liczby całkowitej Pythona 2.x. Z tego powodu liczby całkowite nie mogą już być zapisywane z końcową literą l lub L, a liczby tego typu nigdy nie są również wyświetlane z tymi znakami. Poza tym ograniczeniem opisana zmiana nie wpłynie na większość programów, o ile nie wykonują one sprawdzania typów pod kątem długich liczb całkowitych z wersji 2.x.

Literały szesnastkowe, ósemkowe oraz dwójkowe

Liczby całkowite mogą być w Pythonie zapisane jako dziesiętne (o podstawie dziesięć), szesnastkowe (o podstawie szesnaście), ósemkowe (o podstawie osiem) oraz dwójkowe (o podstawie dwa), z których trzy ostatnie są wspólne w niektórych domenach programistycznych. Literały szesnastkowe zaczynają się od znaków 0x lub 0X, po których następuje ciąg cyfr szesnastkowych od 0 do 9 i od A do F. W literałach szesnastkowych cyfry szesnastkowe mogą być zapisane małą i wielką literą. Literały ósemkowe rozpoczynają się od znaków 0o lub 0O (cyfry zero i małej lub wielkiej litery „o”), po których następuje ciąg cyfr od 0 do 7. W Pythonie 2.x i wcześniejszych wersjach literały ósemkowe mogły także być zapisywane z samą początkową cyfrą 0, jednak od wersji 3.x jest to niemożliwe (początkowy format liczby ósemkowej można łatwo pomylić z liczbą dziesiętną, dlatego zastąpiono go nowym formatem 0o). Literały dwójkowe, stanowiące nowość w wersjach 2.6 i 3.0, rozpoczynają się od znaków 0b lub 0B, po których następują cyfry dwójkowe (0 lub 1).

Warto zauważyć, że wszystkie powyższe literały tworzą w kodzie programu obiekty liczb całkowitych, które są tylko alternatywnymi sposobami zapisu określonych wartości. Wywołania wbudowanych funkcji hex(*liczba całkowita*), oct(*liczba całkowita*) oraz bin(*liczba całkowita*) przekształcają liczbę całkowitą na jej reprezentację o podanej podstawie, natomiast wywołanie int(*str, podstawa*) przekształca wykonywany łańcuch na liczbę całkowitą zgodnie z podaną podstawą.

Liczby zespolone

Literały liczb zespolonych w Pythonie zapisywane są jako część_rzeczywista+część_urojona, gdzie część_urojona kończy się znakiem j lub J. Fragment części_rzeczywista jest opcjonalny, dlatego część_urojona może występować samodzielnie. Wewnętrznie liczby zespolone zaimplementowane są jako pary liczb zmiennoprzecinkowych, jednak wszystkie operacje liczbowe na liczbach zespolonych wykorzystują arytmetykę liczb zespolonych. Liczby zespolone można również tworzyć za pomocą wywołania wbudowanej funkcji complex(część_rzeczywista, część_urojona).

Inne typy liczbowe

Jak zobaczymy w dalszej części niniejszego rozdziału, istnieją jeszcze inne typy liczbowe, spełniające bardziej zaawansowane lub wyspecjalizowane role, które zostały wymienione w końcowej części tabeli 5.1. Część z nich tworzona jest za pomocą wywoływania funkcji z importowanych modułów (na przykład liczby dziesiętne oraz ułamkowe), natomiast inne posiadają własną składnię literałów (na przykład zbiory).

Wbudowane narzędzia liczbowe

Oprócz zaprezentowanych w tabeli 5.1 wbudowanych literałów liczbowych i wywołań konstruktorów Python udostępnia również zbiór narzędzi służących do przetwarzania obiektów liczbowych:

Operatory wyrażeń

`+, -, *, /, >>, **, & i inne`

Wbudowane funkcje matematyczne

`pow, abs, round, int, hex, bin i inne`

Moduły narzędziowe

`random, math i inne`

Spotkamy się z każdym z nich w miarę omawiania kolejnych zagadnień w tym rozdziale.

Choć liczby przetwarzają się przede wszystkim za pomocą wyrażeń, wbudowanych funkcji oraz modułów, udostępniają one również obecnie pewną liczbę *metod* specyficznych dla poszczególnych typów, które także zostaną omówione w niniejszym rozdziale. Liczby zmienoprzecinkowe mają na przykład metodę `as_integer_ratio` przydającą się dla typu liczby ułamkowej, a także metodę `is_integer` sprawdzającą, czy liczba ta jest całkowita. Liczby całkowite dysponują różnymi atrybutami, w tym nową metodą `bit_length`, wprowadzoną w wersji 3.1 Pythona, która podaje liczbę bitów niezbędną do odzwierciedlenia wartości obiektu. Co więcej, *zbiory*, będące po części typem kolekcji, a po części typem liczbowym, także obsługują zarówno metody, jak i wyrażenia.

Ponieważ to wyrażenia są najważniejszym narzędziem w przypadku większości typów liczbowych, to nasze rozważania rozpocznimy właśnie od nich.

Operatorы wyrażeń Pythona

Chyba najważniejszym narzędziem przetwarzającym liczby jest *wyrażenie* (ang. *expression*) — kombinacja liczb (i innych obiektów) oraz operatorów, które po uruchomieniu w Pythonie oblicza wartość. W Pythonie wyrażenia zapisywane są z wykorzystaniem zwykłej notacji matematycznej oraz symboli operatorów. Żeby na przykład dodać do siebie dwie liczby, X i Y, należy stworzyć wyrażenie `X + Y`, które nakazuje Pythonowi zastosować operator `+` do wartości o nazwach X i Y. Wynikiem tego wyrażenia jest suma liczb X oraz Y, czyli inny obiekt liczbowy.

W tabeli 5.2 wymieniono wszystkie wyrażenia z operatorami dostępne w Pythonie. Wiele z nich jest oczywistych; obsługiwane są na przykład zwykłe operatory matematyczne (jak `+`, `-`, `*` czy `/`). Kilka będzie wyglądało znajomo dla osób, które w przeszłości używały innych języków programowania — i tak `%` oblicza resztę z dzielenia, `<<` wykonuje przesunięcie bitowe w lewo, a `&` oblicza wynik dla bitowego AND. Inne są specyficzne dla Pythona i nie wszystkie są z natury numeryczne. Operator `is` sprawdza na przykład tożsamość obiektu (to znaczy adres w pamięci, scisłą formę równości), a `lambda` tworzy funkcje anonimowe.

Tabela 5.2. Operatorы wyrażeń Pythona wraz z priorytetem

Operatorы	Opis
<code>yield x</code>	Protokół send funkcji generatora
<code>lambda argumenty: wyrażenie</code>	Generowanie funkcji anonimowej
<code>x if y else z</code>	Trójargumentowe wyrażenie wyboru (x jest obliczane jedynie wtedy, gdy y jest prawdziwe)
<code>x or y</code>	Logiczne OR (y jest obliczane tylko wtedy, gdy x jest fałszywe)
<code>x and y</code>	Logiczne AND (y jest obliczane tylko wtedy, gdy x jest prawdziwe)
<code>not x</code>	Logiczna negacja

<code>x in y, x not in y</code>	Przynależność (obiekty, po których można iterować, zbiory)
<code>x is y, x is not y</code>	Testy identyczności obiektów
<code>x < y, x <= y, x > y, x >= y x == y, x != y</code>	Operatory porównania, podzbiory i nadzbiory zbiorów; Operatory równości wartości
<code>x y</code>	Bitowe OR, suma zbiorów
<code>x ^ y</code>	Bitowe XOR, symetryczna różnica zbiorów
<code>x & y</code>	Bitowe AND, część wspólna zbiorów
<code>x << y, x >> y</code>	Przesunięcie x w lewo lub prawo o y bitów
<code>x + y x - y</code>	Dodawanie, konkatenacja; Odejmowanie, różnica zbiorów
<code>x * y x % y x / y, x // y</code>	Mnożenie, powtórzenie; Reszta z dzielenia, format; Dzielenie: prawdziwe i bez reszty
<code>-x, +x</code>	Negacja, identyczność
<code>~x</code>	Bitowe NOT
<code>x ** y</code>	Potęga
<code>x[i]</code>	Indeksowanie (sekwencje, odwzorowania, inne)
<code>x[i:j:k]</code>	Wycinki
<code>x(...)</code>	Wywołanie (funkcji, metod, klasy, innych obiektów, które można wywoływać)
<code>x.atrybut,</code>	Odrowlanie do atrybutu
<code>(...)</code>	Krotka, wyrażenie, wyrażenie generatora
<code>[...]</code>	Lista, lista składana
<code>{...}</code>	Słownik, zbiór, a także zbiór i słownik składany

Ponieważ w niniejszej książce omawiamy Pythona w wersjach 2.x oraz 3.x, poniżej znajduje się kilka uwag dotyczących różnic między tymi wersjami, a także nowych dodatków odnoszących się do operatorów wymienionych w tabeli 5.2.

- W Pythonie 2.x nierówność wartości można zapisać albo jako `X != Y`, albo `X <> Y`. W Pythonie 3.x druga z tych opcji została usunięta, ponieważ jest zbędna. W obu wersjach zalecane jest używanie w testach nierówności zapisu `X != Y`.
- W Pythonie 2.x wyrażenie z apostrofem odwrotnym `X` działa tak samo jak funkcja `repr(X)` i przekształca obiekty na wyświetlanełańcuchy znaków. Ze względu na niejasność wyrażenie to zostało w Pythonie 3.x usunięte. Teraz należy korzystać z o wiele bardziej czytelnych funkcji `str` oraz `repr`, opisanych w podrozdziale „Formaty wyświetlania liczb” w niniejszym rozdziale.

- Operator dzielenia bez reszty (ang. *floor division*) $X // Y$ zawsze odcina ułamkową resztę z dzielenia — zarówno w Pythonie 2.x, jak i 3.x. Wyrażenie X / Y wykonuje w wersji 3.x prawdziwe dzielenie (zachowując resztę), natomiast w wersji 2.x — dzielenie klasyczne (odcinanie reszty do liczby całkowitej). Więcej informacji na ten temat znajdziesz w podrozdziale „Dzielenie — klasyczne, bez reszty i prawdziwe”.
- Składnia `[...]` może reprezentować albo literał listy, albo wyrażenie list składanych. Ta druga opcja wykonuje pętlę i zbiera wynik wyrażenia do nowej listy. Przykłady takiego działania można znaleźć w rozdziałach 4., 14. oraz 20.
- Składnia `(...)` wykorzystywana jest w krotkach, grupowaniu wyrażeń, a także w wyrażeniach generatorów — formie list składanych zwracającej wyniki na żądanie zamiast budowania listy wyników. Przykłady znajdują się w rozdziałach 4. oraz 20. We wszystkich trzech konstrukcjach można czasami pominąć nawiasy. W przypadku pominięcia nawiasów krotki `przecinek` oddzielający jej elementy działa jak operator o najniższym priorytecie, o ile nie ma innego znaczenia.
- Składnia `{ ... }` wykorzystywana jest w literałach słowników, natomiast w Pythonie 3.x i 2.7 w literałach zbiorów i słownikach oraz zbiorach składanych. Przykłady znajdziesz w omówieniu zbiorów w tym rozdziale oraz w rozdziałach 4., 8., 14. i 20.
- Wyrażenia `yield` oraz trójargumentowe wyrażenie wyboru `if/else` dostępne są w Pythonie w wersji 2.5 i nowszych. Pierwsze z nich zwraca argumenty `send(...)` w generatorach. Drugie jest skrótem dla wielowierszowej instrukcji `if`. Wyrażenie `yield` wymaga użycia nawiasów, jeżeli nie znajduje się po prawej stronie instrukcji przypisania jako jedyny element.
- Operatory porównania można ze sobą łączyć. Kod $X < Y < Z$ zwraca ten sam wynik co $X < Y$ i $Y < Z$. Więcej informacji na ten temat znajdziesz w podrozdziale „Porównania — zwykłe i łączone” w niniejszym rozdziale.
- W nowszych wersjach Pythona wyrażenie wycinka `X[I:J:K]` jest równoważne z indeksowaniem z obiektem wycinka `X[slice(I, J, K)]`.
- W wersjach 2.x dozwolone jest porównywanie wielkości obiektów o mieszanych typach, konwersja liczb na wspólny typ i porządkowanie innych typów mieszanych zgodnie z nazwą typu. W Pythonie 3.x porównywanie wielkości obiektów nieliczbowych o mieszanych typach nie jest dozwolone i powoduje zgłoszenie wyjątków. Obejmuje to także sortowanie za pomocą metody pośredniczącej.
- W Pythonie 3.x nie są już również obsługiwane porównania wielkości słowników (choć testy równości są). Jedynym dostępnym rozwiązaniem pozostaje porównanie `sorted(dict.items())`.

Większość operatorów z tabeli 5.2 zobaczymy w akcji nieco później. Najpierw musimy jednak przyjrzeć się sposobom łączenia tych operatorów w wyrażeniach.

Połączone operatory stosują się do priorytetów

Tak jak w większości języków programowania, również w Pythonie bardziej skomplikowane wyrażenia kodujemy, łącząc ze sobą wyrażenia z operatorami z tabeli 5.2. Sumę dwóch operacji mnożenia można na przykład zapisać jak poniższe połączenie zmiennych i operatorów.

`A * B + C * D`

Skąd zatem Python wie, jakie działania ma wykonać jako pierwsze? Odpowiedź na to pytanie leży w *priorytecie operatorów* (ang. *operator precedence*). Kiedy piszemy wyrażenie zawierające więcej niż jeden operator, Python grupuje jego części zgodnie z tak zwanymi *regułami precedencji*. To grupowanie ustala kolejność obliczania poszczególnych części wyrażenia. Tabela 5.2 została posortowana zgodnie z priorytetem operatorów:

- Operatory znajdujące się niżej w tabeli mają wyższy priorytet, dlatego wiążą bardziej ściśle w wyrażeniach mieszanych.
- Operatory znajdujące się w tym samym wierszu tabeli 5.2 zazwyczaj po połączeniu grupują od lewej do prawej strony (z wyjątkiem potęgowania, które grupuje od prawej do lewej strony, i porównań, łączących w łańcuchy od lewej do prawej strony).

Jeżeli na przykład zapiszemy wyrażenie $X + Y * Z$, Python najpierw wykona mnożenie ($Y * Z$), którego wynik doda następnie do X , ponieważ $*$ ma wyższy priorytet (jest niżej w tabeli) od $+$. W podobny sposób w pierwszym przykładzie najpierw wykonane zostaną oba działania mnożenia ($A * B$ oraz $C * D$), a dopiero później ich wyniki zostaną do siebie dodane.

Podwyrażenia grupowane są w nawiasach

O priorytecie operatorów można zapomnieć, jeżeli będzie się uważnie grupowało części wyrażeń w nawiasy. Kiedy podwyrażenia umieści się w nawiasach, będą one ważniejsze od reguł priorytetów Pythona. Python zawsze najpierw wykonuje wyrażenia znajdujące się w nawiasach, a dopiero później ich wyniki wykorzystuje w zawierających je wyrażeniach.

Zamiast na przykład pisać $X + Y * Z$, można zapisać to wyrażenie na dwa sposoby, zmuszając jednocześnie Pythona do obliczenia go w pożądanej kolejności.

```
(X + Y) * Z  
X + (Y * Z)
```

W pierwszym przypadku najpierw dodawane są do siebie elementy X i Y , ponieważ to wyrażenie umieszczone jest w nawiasach. W drugim przypadku najpierw wykonywane jest mnożenie (dokładnie tak samo by było, gdyby nawiasy zostały pominięte). Dodawanie nawiasów w większych wyrażeniach jest dobrym pomysłem. Nie tylko wymusza to odpowiednią kolejność wykonywania działań, ale dodatkowo poprawia również czytelność.

Pomieszane typy poddawane są konwersji

Poza mieszaniem operatorów w wyrażeniach można również łączyć ze sobą typy liczbowe. Można na przykład dodać liczbę całkowitą do liczby zmiennoprzecinkowej.

```
40 + 3.14
```

Prowadzi to jednak do kolejnego pytania: jakiego typu będzie wynik takiego działania — będzie liczbą całkowitą czy zmiennoprzecinkową? Odpowiedź jest prosta, szczególnie dla osób, które używały już kiedyś jakiegoś języka programowania. W wyrażeniach z mieszanymi typami obiektów liczbowych Python najpierw dokonuje konwersji argumentów *w górę*, do typu, jakim jest najbardziej skomplikowany argument, a dopiero później wykonuje obliczenia dla argumentów tego samego typu. Osoby korzystające kiedyś z języka C pamiętają zapewne, że podobne konwersje typów zastosowano także w tym języku.

Python mierzy stopień skomplikowania typów liczbowych w następującej kolejności: liczby całkowite są prostsze od liczb zmiennoprzecinkowych, które są prostsze od liczb zespolonych. Kiedy zatem liczba całkowita zostanie pomieszana ze zmiennoprzecinkową, jak w przykładzie wyżej, zostanie ona najpierw przekonwertowana na wartość zmiennoprzecinkową i w tym samym formacie zwrotnie zostanie wynik działania.

```
>>> 40 + 3.14 # Zamiana liczby całkowitej na zmiennoprzecinkową, obliczenia i wynik w postaci
```

```
# zmiennoprzecinkowej
```

```
43.14
```

Podobnie każde wyrażenie typu mieszanego, w którym jeden operand jest liczbą zespoloną, powoduje przekształcenie drugiego operandu w liczbę zespoloną i daje wynik będący liczbą zespoloną. W Pythonie 2.x zwykłe liczby całkowite przekształcane są także na długie liczby całkowite w każdej sytuacji, kiedy ich wartość jest zbyt duża, by zmieścić się w zwykłej liczbie całkowitej. W wersji 3.x liczby całkowite wchodzące długie liczby całkowite.

Takie zachowanie można również wymusić, ręcznie wywołując wbudowane funkcje służące do konwersji typów.

```
>>> int(3.1415)          # Odcina liczbę zmiennoprzecinkową do całkowitej  
3  
>>> float(3)           # Przekształca liczbę całkowitą na  
zmiennoprzecinkową  
3.0
```

Zazwyczaj jednak nie będzie trzeba tego robić. Ponieważ Python automatycznie konwertuje typy w górę do najbardziej złożonego typu z danego wyrażenia, wynik będzie na ogół w odpowiedniej postaci.

Należy również pamiętać, że wszystkie konwersje mieszanych typów mają miejsce tylko przy łączeniu typów *liczbowych* (na przykład liczby całkowitej i zmiennoprzecinkowej) w wyrażeniach, w tym wyrażeniach wykorzystujących operatory liczbowe i porównania. Generalnie Python nie dokonuje automatycznej konwersji w przypadku typów innego rodzaju. Przykładowo dodanie łańcucha znaków do liczby całkowitej spowoduje wyświetlenie błędu, o ile ręcznie nie przekształcimy jednego z nich. Przykład takiego działania będzie można znaleźć przy okazji omawiania łańcuchów znaków w rozdziale 7.



W Pythonie 2.X można porównywać nieliczbowe typy mieszane, ale nie są wykonywane żadne konwersje — typy mieszane są porównywane zgodnie z regułą, która wydaje się deterministyczna, ale nie estetyczna: porównywane są nazwy typów obiektów. W wersji 3.x porównywanie wielkości nieliczbowych typów mieszanych nie jest dozwolone i powoduje zgłoszenie wyjątków. Zauważ, że dotyczy to tylko operatorów porównania, takich jak `>`; inne operatory, takie jak `+`, nie zezwalają na mieszanie typów nieliczbowych w wersjach 3.x i 2.x.

Wprowadzenie: przeciążanie operatorów i polimorfizm

Choć obecnie skupiamy się na wbudowanych typach liczbowych, należy pamiętać, że w Pythonie wszystkie operatory mogą być przeciążane (to znaczy implementowane) przez klasy Pythona oraz typy z rozszerzeń języka C, tak by działały na tworzonych obiektach. Później zobaczymy na przykład, że obiekty utworzone za pomocą klas można dodawać i poddawać konkatenacji za pomocą wyrażeń `x+y`, indeksowaniu za pomocą wyrażeń `x[i]` i tak dalej.

Co więcej, Python sam automatycznie przeciąża niektóre operatory, tak by wykonywały one różne zadania w zależności od typu przetwarzanego obiektu wbudowanego. Operator `+` wykonuje na przykład operację dodawania, kiedy zastosuje się go na liczbach, ale w połączeniu z łańcuchami znaków oraz listami odpowiedzialny jest za konkatenację. Operator `+` może oznaczać dowolne działania po zastosowaniu do obiektów zdefiniowanych za pomocą klas.

Jak widzieliśmy w poprzednim rozdziale, właściwość ta nazywana jest najczęściej *polimorfizmem*. Termin ten oznacza, że znaczenie operacji uzależnione jest od typu obiektów, na których się tę operację wykonuje. Koncepcję tą zajmiemy się ponownie przy okazji omawiania funkcji w rozdziale 16., ponieważ w tym kontekście stanie się ona o wiele bardziej oczywista.

Liczby w akcji

Czas przejść do kodu! Chyba najlepszym sposobem zrozumienia obiektów liczbowych i związanych z nimi wyrażeń jest zobaczenie ich w akcji, więc mając już ugruntowane podstawy, możesz uruchomić interaktywny wiersz poleceń i przetestować kilka podstawowych operacji (wskazówki dotyczące uruchamiania sesji interaktywnej znajdziesz w rozdziale 3.).

Zmienne i podstawowe wyrażenia

Zacznijmy od podstawowej matematyki. W poniższej sesji interaktywnej najpierw przypiszemy dwie *zmienne* (*a* oraz *b*) do liczb całkowitych, tak by móc z nich później skorzystać w większym wyrażeniu. Zmienne to po prostu nazwy utworzone przez Pythona, które wykorzystywane są do śledzenia informacji w programie.Więcej na ten temat powiemy w kolejnym rozdziale, jednak na razie warto podkreślić, że w Pythonie:

- Zmienne tworzone są, kiedy pierwszy raz przypisze się do nich wartości.
- Zmienne zastępowane są wartościami, kiedy wykorzystywane są w wyrażeniach.
- Zmienne muszą być przypisane przed użyciem w wyrażeniach.
- Zmienne odnoszą się do obiektów i nigdy nie są wcześniej deklarowane.

Innymi słowy, przypisania te sprawią, że zmienne *a* i *b* zaistnieją automatycznie.

```
% python
>>> a = 3                      # Utworzenie zmiennej (nazwy); zmienne nie są
wcześniej deklarowane
>>> b = 4
```

W kodzie powyżej wykorzystano również *komentarz*. Warto przypomnieć, że w kodzie napisanym w Pythonie tekst od znaku `#` aż do końca wiersza jest uznawany za komentarz i tym samym ignorowany przez interpreter. Komentarze to metoda zapisu w kodzie programu dokumentacji czytelnej dla użytkownika i jednocześnie ważny element procesu tworzenia oprogramowania. Pisząc tę książkę, dodałem komentarze w większości przykładów z książki w celu ułatwienia zrozumienia kodu. W dalszej części książki poznasz pokrewny, ale znacznie bardziej funkcjonalny mechanizm — tzw. notki dokumentacyjne (ang. *documentation strings*) lub w skrócie notki *docstrings*, które dołączają tekst komentarzy do obiektów.

Ponieważ kod wpisywany interaktywnie jest tymczasowy, zazwyczaj w tym kontekście nie stosuje się komentarzy. Jeżeli podczas pracy z książką próbujesz samodzielnie wykonywać wszystkie opisywane przykłady, to pamiętaj, że nie musisz wpisywać tekstów komentarzy od znaku `#` do końca wiersza, ponieważ nie są one częściami poleceń, które uruchamiamy.

Teraz wykorzystamy nasze nowe obiekty liczb całkowitych w wyrażeniach. W tym momencie zmienne *a* i *b* będą miały wartości odpowiednio 3 i 4. Zmienne tego typu są zastępowane wartościami za każdym razem, gdy znajdują się w wyrażeniu. W czasie pracy w sesji interaktywnej wyniki wyrażenia są zwracane od razu.

```
>>> a + 1, a - 1                  # Dodawanie (3 + 1), odejmowanie (3 - 1)
(4, 2)
>>> b * 3, b / 2                  # Mnożenie (4 * 3), dzielenie (4 / 2,
wynik dla wersji 3.x)
(12, 2)
>>> a % 2, b ** 2                # Modulo (reszta z dzielenia),
potęgowanie (4 ** 2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b            # Konwersja typów mieszanych
(6.0, 16.0)
```

Z technicznego punktu widzenia zwracane wyniki są *krotkami* składającymi się z dwóch wartości, ponieważ wiersze wpisywane w sesji interaktywnej zawierają po dwa wyrażenia rozdzielone przecinkiem — dlatego właśnie wyniki wyświetlane są w nawiasach (więcej o

krotkach w kolejnych rozdziałach). Warto zauważać, że wyrażenia działają, ponieważ znajdują się w nich zmienne `a` i `b` mają przypisane wartości. Gdybyśmy użyli innej zmiennej, do której *nie zostały przypisane żadne wartości*, próba wykonania takiego wyrażenia zakończyłaby się błędem, ponieważ Python nie przypisuje do nieistniejących zmiennych żadnych wartości domyślnych.

```
>>> c * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    NameError: name 'c' is not defined
```

Zmiennych nie trzeba w Pythonie wcześniej deklarować, jednak przed użyciem trzeba im przynajmniej raz przypisać jakąś wartość. W praktyce oznacza to, że liczniki trzeba inicjalizować z wartością zero, zanim będzie można coś do nich dodawać; w podobny sposób przed dodaniem czegoś do listy najpierw trzeba zainicjować pustą listę i tak dalej.

Poniżej znajdują się dwa nieco bardziej rozbudowane wyrażenia ilustrujące grupowanie operatorów, konwersje typów i różnice w działaniu operatora dzielenia w wersjach 2.x i 3.x Pythona.

```
>>> b / 2 + a          # To samo co ((4 / 2) + 3) [w wersji 2.x użyj 2.0]
5.0
>>> b / (2.0 + a)      # To samo co (4 / (2.0 + 3)) [w wersjach
wcześniejszych niż 2.7 użyj funkcji print]
0.8
```

W pierwszym wyrażeniu nie ma nawiasów, dlatego Python automatycznie grupuje argumenty zgodnie z regułami priorytetów. Ponieważ operator `/` jest w tabeli 5.2 niżej od `+`, ma wyższy priorytet jest obliczany jako pierwszy. Z tego powodu całosć obliczana jest w taki sam sposób, jakby wyrażenie zawierało nawiasy takie jak w komentarzu.

Warto również zauważać, że wszystkie liczby z pierwszego wyrażenia są *liczbami całkowitymi*. Z tego powodu Python 2.x wykonuje operacje dzielenia i dodawania liczb całkowitych i zwróci wynik 5, podczas gdy Python 3.x wykonuje prawdziwe dzielenie z resztą i zwraca wynik 5.0. Jeżeli chcesz w wersji 3.x uzyskać dzielenie liczb całkowitych, takie jak w wersji 2.x, powinieneś zapisać powyższe działanie w postaci `b // 2 + a`; jeżeli w wersji 2.x chcesz uzyskać prawdziwe dzielenie z wersji 3.x, powinieneś zapisać nasze wyrażenie w następujący sposób: `b / 2.0 + a` (więcej informacji na temat dzielenia już niebawem).

W drugim wyrażeniu nawiasy dodawane są wokół części z operatorem `+` w celu zmuszenia Pythona do obliczenia jej na początku (przed operatorem `/`). Jeden z argumentów staje się również liczbą zmiennoprzecinkową ze względu na dodanie do wyrażenia liczby 2.0. Ponieważ wyrażenie zawiera mieszane typy, Python konwertuje liczbę całkowitą kryjącą się pod zmienną `a` na wartość zmiennoprzecinkową (3.0), jeszcze zanim wykona dodawanie. Gdyby wszystkie liczby w tym wyrażeniu były liczbami całkowitymi, w rezultacie dzielenia liczb całkowitych (4 / 5) w Pythonie 2.x wynik zostałby odcięty do 0, natomiast w Pythonie 3.x wynik będzie liczbą zmiennoprzecinkową 0.8 (więcej informacji o dzieleniu już wkrótce).

Formaty wyświetlania liczb

Jeżeli używasz języka Python w wersji 2.6, 3.0 lub wcześniejszej, wynik działania ostatniego wyrażenia z poprzedniego przykładu może na pierwszy rzut oka wyglądać nieco dziwnie:

```
>>> b / (2.0 + a)          # w wersjach <=2.6: wyniki działania
zawierają więcej (lub mniej) cyfr...
```

```
0.8000000000000000
>>> print(b / (2.0 + a))          # ... ale funkcja print zaokrąglą liczbę
0.8
```

Przyczyny takiego działania Pythona objaśnialiśmy krótko w poprzednim rozdziale i nie występuje już ono w Pythonie 2.7, 3.1 i nowszych. Pełne wyjaśnienie tego dziwnego wyniku jest związane ze sprzętowymi ograniczeniami procesorów przeprowadzających obliczenia zmiennoprzecinkowe, które nie są w stanie przedstawić dokładnych wartości niektórych wyrażeń w ograniczonej liczbie bitów. Ponieważ jednak sprzętowa architektura komputerów znacznie wykracza poza tematykę niniejszej książki, wystarczy powiedzieć, że układy zmiennoprzecinkowe Twojego komputera robią wszystko, co w ich mocy, i ani one, ani Python nie popełniają tutaj żadnego błędu.

W rzeczywistości jest to tylko niewielki problem z *wyświetlaniem* na ekranie — automatyczne wyświetlanie wyników w sesji interaktywnej pokazuje więcej cyfr niż funkcja `print` tylko dlatego, że po prostu używa do tego celu nieco innego algorytmu, a w pamięci komputera jest to wciąż jedna i ta sama liczba. Jeżeli nie chcesz widzieć tych wszystkich cyfr, powinieneś skorzystać z funkcji `print`. Zgodnie z informacjami zamieszczonymi w ramce „Formaty wyświetlania str i repr”, którą znajdziesz w dalszej części tego rozdziału, właśnie tak można uzyskać sposób wyświetlania bardziej przyjazny dla użytkownika. Począwszy od wersji 2.7 i 3.1, zmiennoprzecinkowa logika wyświetlania Pythona jest już nieco bardziej inteligentna i zwykle pokazuje mniej cyfr dziesiętnych (choć czasami zdarza się jej wyświetlić ich więcej).

Warto jednak zauważyć, że nie wszystkie wartości zawierają tyle cyfr do wyświetlenia.

```
>>> 1 / 2.0
0.5
```

Istnieje więcej sposobów prezentowania wartości liczbowych niż tylko używanie funkcji `print` i automatyczne wyświetlanie wyników w sesji interaktywnej:

```
>>> num = 1 / 3.0
>>> num                      # Automatyczne wyświetlanie wyniku
0.3333333333333333
>>> print(num)                # Dokładne wyświetlanie liczby
0.3333333333333333
>>> '%e' % num                # Formatowanie za pomocą łańcuchów znaków
'3.33333e-01'
>>> '%.4.2f' % num             # Alternatywne formatowanie liczb
zmiennoprzecinkowych
'0.33'
>>> '{0:4.2f}'.format(num)     # Metoda formatowania za pomocą łańcuchów
znaków (Python 2.6, 3.0 i nowsze)
'0.33'
```

Formaty wyświetlania — funkcje `str()` i `repr()`

Z technicznego punktu widzenia różnica między automatycznym wyświetlaniem z sesji interaktywnej a funkcją `print` odpowiada różnicy między wbudowanymi funkcjami `str` i `repr`.

```
>>> repr('spam')           # Wyświetlanie w postaci jak w kodzie (sesja
   interaktywna)
" 'spam' "
>>> str('spam')            # Wyświetlanie w formie przyjaznej dla
   użytkownika (odpowiednik print)
'spam'
```

Obie funkcje konwertują dowolne obiekty na ich reprezentacje łańcuchowe. Funkcja `repr` (i domyślne wyświetlanie z sesji interaktywnej) zwraca wyniki wyglądające tak, jakby były one kodem. Funkcja `str` (oraz operacja `print`) dokonuje konwersji na format bardziej przyjazny dla użytkownika, jeśli jest on dostępny. Niektóre obiekty mają obie postacie — `str` do użytku ogólnego i `repr` z dodatkowymi szczegółami. Koncepcje te pojawią się ponownie przy okazji omawiania łańcuchów znaków oraz przeciążania operatorów w klasach, a więcej informacji na temat tych funkcji wbudowanych znajduje się w dalszej części książki.

Oprócz udostępniania wyświetlanych łańcuchów znaków dla dowolnych obiektów wbudowana funkcja `str` jest także nazwą typu danych łańcucha znaków i w wersji 3.x można ją wywołać z nazwą schematu kodowania w celu zdekodowania łańcucha znaków Unicode z ciągu bajtowego (na przykład `str(b'xy', 'utf8')`, co stanowi alternatywę dla wywołania metody `bytes.decode`, którą poznaliśmy w rozdziale 4. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 37).

Ostatnie trzy wyrażenia wykorzystują *formatowanie za pomocą łańcucha znaków* — jest to narzędzie pozwalające na elastyczne formatowanie elementów wyświetlanych na ekranie, które omówimy w rozdziale 7. poświęconym łańcuchom znaków. Wynikiem takiego formatowania są łańcuchy znaków, które są zazwyczaj wyświetlane na ekranie lub zapisywane w raportach.

Porównania — zwykłe i łączone

Dotychczas zajmowaliśmy się standardowymi działaniami na liczbach (dodawaniem czy mnożeniem), ale liczby, podobnie jak wszystkie inne obiekty w Pythonie, można także porównywać. Zwykłe porównania działają w przypadku liczb tak, jak można by się tego spodziewać — porównują względną wielkość argumentów i zwracają wynik w postaci wartości Boolean, którą zazwyczaj sprawdzamy w kodzie programu i w zależności od wyniku testu podejmujemy odpowiednie działania.

```
>>> 1 < 2                  # Mniejszy od
True
>>> 2.0 >= 1              # Większy od bądź równy — mieszane typy, więc 1
   przekształcone zostaje na 1.0
True
>>> 2.0 == 2.0             # Sprawdzenie, czy wartości są równe
True
>>> 2.0 != 2.0             # Sprawdzenie, czy wartości nie są równe
False
```

Warto raz jeszcze zwrócić uwagę na to, że w wyrażeniach liczbowych (i tylko tam) można używać typów mieszanych. W drugim teście Python porównuje wartości jako obiekty typu o większej złożoności, czyli liczby zmiennoprzecinkowe.

Co ciekawe, Python pozwala także łączyć ze sobą większą liczbę porównań w celu wykonywania testów przedziałów. Połączone porównania są w pewnym sensie skrótem dla większych wyrażeń Boolean. W skrócie, Python pozwala połączyć ze sobą kilka testów porównania wielkości w celu utworzenia kodu porównań łączonych, takich jak testy przedziałów. Wyrażenie ($A < B < C$) sprawdza na przykład, czy B znajduje się pomiędzy A i C , i jest równoznaczne z testem Boolean ($A < B$ and $B < C$), jednak przyjemniejsze dla oka (i klawiatury). Założymy, że mamy poniższe instrukcje przypisania:

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

Dwa zaprezentowane niżej wyrażenia dają identyczny efekt, jednak pierwsze z nich jest krótsze i może działać nieco szybciej, ponieważ Python musi obliczyć Y tylko raz:

```
>>> X < Y < Z          # Połączone porównania – testy
przedziału
True
>>> X < Y and Y < Z
True
```

Tak samo równoznaczne są testy zwracające wynik `False`. Dozwolone są również łańcuchy porównań o dowolnej długości:

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False
>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

W połączonych testach można także skorzystać z innych porównań, jednak niektóre wyrażenia zapisane w ten sposób mogą być nieintuicyjne, o ile nie obliczamy ich w ten sam sposób co Python. Poniższe wyrażenie zwraca na przykład `False` tylko dlatego, że 1 nie jest równe 2:

```
>>> 1 == 2 < 3          # To samo co: 1 == 2 i 2 < 3
False                     # Nie to samo co: False < 3 (co
                           oznacza, że 0 < 3, co jest prawdą!)
```

Python nie porównuje wyniku `False` z wyrażeniem `1 == 2` do 3 — to akurat oznaczałoby to samo co $0 < 3$, czyli powinno zwracać wartość `True` (jak zobaczymy w dalszej części niniejszego rozdziału, `True` i `False` są po prostu zapisanymi w innym formacie liczbami 1 oraz 0).

Jeszcze jedna uwaga, zanim przejdziemy dalej: porównania numeryczne oparte są na wielkościach, które są na ogół proste — chociaż liczby zmienoprzecinkowe nie zawsze działają tak, jak można się spodziewać, i mogą wymagać uprzedniej konwersji lub innego dostosowania:

```
>>> 1.1 + 2.2 == 3.3      # Czy to przypadkiem nie powinna być prawda
                           (True)?
```

```

False

>>> 1.1 + 2.2                      # Wynik to prawie 3.3, ale nie dokładnie;
problemem jest ograniczona          # precyzja liczb
                                         # trunc w kolejnych
                                         # podrozdziałach

True                                     # Ułamki zwykłe i dziesiętne również mogą
                                         # tutaj pomóc
                                         # (zobacz
                                         # kolejne podrozdziały)

```

Wynika to z faktu, że za pomocą liczb zmiennoprzecinkowych nie możemy dokładnie odwzorować niektórych wartości ze względu na ograniczoną liczbę bitów — to podstawowy problem w programowaniu numerycznym, który nie dotyczy tylko Pythona; więcej szczegółowych informacji na ten temat znajdziesz w kolejnych rozdziałach, gdzie spotkamy ułamki dziesiętne i zwykłe oraz poznamy narzędzia, które pomogą nam rozwiązać takie problemy. Najpierw jednak kontynuujmy naszą prezentację podstawowych operacji numerycznych Pythona, z dokładniejszym omówieniem operacji dzielenia.

Dzielenie – klasyczne, bez reszty i prawdziwe

Widziałeś już w poprzednich podrozdziałach, jak działa dzielenie, zatem powinieneś również wiedzieć, że działanie to zachowuje się nieco inaczej w wersjach 3.x i 2.x. Tak naprawdę istnieją trzy rodzaje dzielenia i dwa różne operatory dzielenia (z czego jeden zmienił się w Pythonie 3.x). Cała historia jest dosyć złożona, ale jest to kolejna poważna zmiana wprowadzona w wersji 3.x, która może spowodować niepoprawne działanie kodu z wersji 2.X, więc warto bardziej szczegółowo wyjaśnić działanie operatora dzielenia:

X / Y

Dzielenie *klasyczne* i *prawdziwe*. W Pythonie 2.x operator ten wykonuje dzielenie *klasyczne*, odcinając resztę dla liczb całkowitych i utrzymując resztę (część ułamkową) dla liczb zmiennoprzecinkowych. W Pythonie 3.x operator ten wykonuje dzielenie *prawdziwe*, które zawsze zachowuje resztę dla wyników zmiennoprzecinkowych, niezależnie od typów obiektów.

X // Y

Dzielenie *bez reszty*. Dodane w Pythonie 2.2 i dostępne w wersjach 2.x oraz 3.x. Operator ten zawsze odcina ułamkową resztę, bez względu na typ obiektu. Rodzaj wyniku zależy od typów operandów.

Prawdziwe dzielenie zostało dodane w celu zniwelowania tego, że wyniki dzielenia klasycznego uzależnione są od typu argumentu, przez co mogą być trudne do przewidzenia w języku z typami dynamicznymi, jakim jest Python. Dzielenie klasyczne zostało usunięte w wersji 3.x z uwagi na to właśnie ograniczenie — operatory / oraz // implementują w tej wersji Pythona dzielenie prawdziwe oraz dzielenie bez reszty. Python 2.x domyślnie ma podział na dzielenie klasyczne i dzielenie bez reszty, ale możesz włączyć dzielenie prawdziwe jako opcję. Podsumowując:

- W wersji 3.x operator / zawsze wykonuje dzielenie *prawdziwe*, zwracając wynik będący liczbą zmiennoprzecinkową uwzględniającą resztę, bez względu na typ argumentów. Operator // wykonuje dzielenie *bez reszty*, odcinające resztę, i zwraca liczbę całkowitą dla argumentów będących liczbami całkowitymi lub liczbą zmiennoprzecinkową, jeśli któryś z argumentów jest liczbą tego typu.
- W wersji 2.x operator / wykonuje dzielenie *klasyczne*, odcinając resztę w przypadku, gdy oba argumenty są liczbami całkowitymi, i zachowując resztę w pozostałych przypadkach. Operator // wykonuje dzielenie *bez reszty* i działa tak samo jak w wersji 3.x, odcinając resztę w przypadku liczb całkowitych i wykonując dzielenie bez reszty w przypadku liczb zmiennoprzecinkowych.

Oto jak działają oba operatory w wersjach 3.x oraz 2.x — pierwsza operacja w każdym zestawie jest kluczową różnicą między wersjami Pythona i może wpływać na działanie kodu:

```
C:\code> C:\Python33\python
>>>
>>> 10 / 4                                # Inaczej niż w 3.x – zachowuje resztę
2.5
>>> 10 // 4.0                               # Tak jak w 3.0 – zachowuje resztę
2.5
>>> 10 / 4                                 # Tak jak w 3.x – odcina resztę
2
>>> 10 // 4.0                               # Tak jak w 3.x – zaokrąglą w dół do
najbliższej liczby całkowitej
2.0
C:\code> C:\Python27\python
>>>
>>> 10 / 4                                # To może spowodować błędne działanie
kodu przeniesionego do 3.x!
2
>>> 10 / 4.0
2.5
>>> 10 // 4                                # Używaj takiego zapisu, jeżeli chcesz
odciąć resztę w 2.x
2
>>> 10 // 4.0
2.0
```

Warto zauważyć, że w wersji 3.x typ danych wyniku dla operatora // w dalszym ciągu uzależniony jest od typu operandów. Jeżeli któryś z nich jest liczbą zmiennoprzecinkową, wynikiem będzie liczba zmiennoprzecinkowa; w przeciwnym wypadku wynik będzie liczbą całkowitą. Choć może się to wydawać podobne do zachowania uzależnionego od typu dla operatora / w wersjach 2.x, który był motywacją dla zmian z wersji 3.x, typ zwracanej wartości jest o wiele mniej istotny od samej zwracanej wartości.

Co więcej, ponieważ operator `//` został udostępniony po części jako narzędzie zapewniające zgodność ze starszymi wersjami w przypadku programów, które opierają się na dzieleniu liczb całkowitych z odcinaniem reszty (co jest o wiele częściej spotykane, niż można by się spodziewać), dla liczb całkowitych musi zwracać liczby całkowite. Zastosowanie operatora `//` zamiast operatora `/` w wersji 2.x, gdy wymagane jest obcięcie reszty z dzielenia, ułatwia utrzymanie kompatybilności kodu z wersją 3.x.

Obsługa różnych wersji Pythona

Choć sposób działania operatora `/` w wersjach 2.x i 3.x różni się od siebie, nadal możesz w kodzie obsługiwać obie wersje Pythona. Jeżeli programy opierają się na dzieleniu liczb całkowitych z odcinaniem reszty, to zarówno w wersji 2.x, jak i 3.x powinieneś użyć operatora `//`. Jeżeli programy wymagają wyników będących liczbami zmiennoprzecinkowymi z resztą dla liczb całkowitych, powinieneś użyć typu `float`, aby zagwarantować, że po wykonaniu programu w wersji 2.x jeden z operandów wyrażenia z operatorem `/` będzie liczbą zmiennoprzecinkową:

```
X = Y // Z          # Zawsze odcina resztę, wynik zawsze jest liczbą
całkowitą dla liczb całkowitych w 2.x i 3.x

X = Y / float(Z)    # Gwarantuje dzielenie liczb zmiennoprzecinkowych z
resztą w 2.x i 3.x
```

Alternatywnie można włączyć dzielenie z operatorem `/` z wersji 3.x w Pythonie 2.x za pomocą wyrażenia `__future__import`, zamiast wymuszać je za pomocą konwersji z użyciem `float`:

```
C:\code> C:\Python27\python

>>> from __future__ import division      # Włączenie obsługi "/" z wersji
3.x

>>> 10 / 4
2.5

>>> 10 // 4                           # Dzielenie liczb całkowitych jest
takie samo w obu wersjach

2
```

Przedstawione wyżej specjalne wyrażenie `from` odnosi się do dalszego ciągu sesji, gdy jest wpisywane interaktywnie, i musi pojawiać się jako pierwszy wykonywalny wiersz kodu, gdy jest używane w pliku skryptu (niestety, jak widać, w Pythonie możemy importować z przyszłości, ale nie z przeszłości... hm, chyba najwyższy czas pogadać z lekarzem...).

Dzielenie bez reszty a odcinanie

Drobny szczegół: operator `//` jest nieformalnie nazywany operatorem *dzielenia z odcinaniem*, jednak bardziej poprawne byłoby nazywanie go *dzieleniem bez reszty* — odcina on resztę, pozostawiając podstawę będącą najbliższą liczbą całkowitą mniejszą od prawdziwego wyniku. W rezultacie zaokrąglamy wynik w dół, a nie odcinamy, co ma znaczenie w przypadku liczb ujemnych. Różnicę tę można zobaczyć na własne oczy, używając modułu Pythona o nazwie `math` (przed użyciem zawartości modułu trzeba go najpierw zaimportować, o czym będziemy mówić nieco później):

```
>>> import math

>>> math.floor(2.5)                  # Najbliższa liczba całkowita mniejsza od
prawdziwego wyniku

2

>>> math.floor(-2.5)
```

```

-3
>>> math.trunc(2.5)          # Odcięcie części ułamkowej (w stronę zera)
2
>>> math.trunc(-2.5)
-2

```

Przy użyciu operatorów dzielenia tak naprawdę odcinamy resztę jedynie dla dodatnich wyników, ponieważ tylko w takim przypadku odcięcie da nam tę samą podstawę. W przypadku liczb ujemnych uzyskujemy w rzeczywistości dzielenie bez reszty (tak naprawdę oba działania są dzieleniem bez reszty, a w przypadku liczb dodatnich dzielenie bez reszty jest równoznaczne z odcinaniem). Oto przykłady z wersji 3.x:

```

C:\code> c:\python33\python
>>> 5 / 2, 5 / -2
(2.5, -2.5)
>>> 5 // 2, 5 // -2      # Odcina resztę do podstawy – zaokrąglą do
                           pierwszej mniejszej liczby całkowitej
(2, -3)                  # 2.5 staje się 2, -2.5 staje się -3
>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)
>>> 5 // 2.0, 5 // -2.0  # Tak samo w przypadku liczb
                           zmiennoprzecinkowych, choć tu wynik jest także
                           # liczbą zmiennoprzecinkową
(2.0, -3.0)

```

W przypadku wersji 2.x jest podobnie, jednak wyniki dla operatora / znów będą różne:

```

C:\code> c:\python27\python
>>> 5 / 2, 5 / -2          # Inaczej niż w 3.x
(2, -3)
>>> 5 // 2, 5 // -2        # Tu i poniżej wyniki w 2.x i 3.x są takie same
(2, -3)
>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)
>>> 5 // 2.0, 5 // -2.0
(2.0, -3.0)

```

Jeżeli naprawdę potrzebne jest nam odcinanie w kierunku zera, bez względu na znak, w każdej wersji Pythona możemy zawsze przekazać wynik dzielenia liczb zmiennoprzecinkowych do funkcji `math.trunc` (powiązaną funkcjonalność można znaleźć we wbudowanej funkcji `round` oraz funkcji `int`, która daje takie same wyniki, ale nie wymaga wcześniejszego zaimportowania):

```

C:\code> c:\python33\python
>>> import math

```

```

>>> 5 / -2                      # Zachowanie reszty
-2.5
>>> 5 // -2                     # Zaokrąglenie wyniku w dół
-3
>>> math.trunc(5 / -2)          # Odcięcie reszty zamiast zaokrąglenia w
dół (tak samo jak w przypadku int())
-2
C:\code> c:\python27\python
>>> import math
>>> 5 / float(-2)               # Reszta w wersji 2.x
-2.5
>>> 5 / -2, 5 // -2            # Zaokrąglenie w dół w wersji 2.x
(-3, -3)
>>> math.trunc(5 / float(-2))   # Odcięcie reszty w wersji 2.x
-2

```

Dlaczego odcinanie ma znaczenie?

Jeżeli korzystasz z Pythona 3.x, poniżej znajdziesz krótkie podsumowanie sposobu działania operatorów dzielenia:

```

>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)           # Prawdziwe
dzielenie w 3.x
(2.5, 2.5, -2.5, -2.5)
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)         # Dzielenie bez
reszty w 3.x
(2, 2.0, -3.0, -3)
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)           # Oba
(3.0, 3.0, 3, 3.0)

```

U osób korzystających z wersji 2.x dzielenie działa następująco (wyniki wyróżnione pogrubieniem różnią się od wyników z wersji 3.x):

```

>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)           # Dzielenie klasyczne w 2.x
(inny wynik niż w 3.x)
(2, 2.5, -2.5, -3)
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)         # Dzielenie bez reszty w
2. (taki sam wynik jak w 3.x)
(2, 2.0, -3.0, -3)
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)           # Oba sposoby dzielenia
(inny wynik niż w 3.x)
(3, 3.0, 3, 3.0)

```

Choć dopiero się o tym przekonamy wraz z upływem czasu, istnieje możliwość, że działanie operatora / bez odcinania reszty w Pythonie 3.x spowoduje problemy w funkcjonowaniu sporej liczby programów napisanych dla wersji 2.x. Być może ze względu na korzenie w składni języka C wielu programistów polega na dzieleniu z odcinaniem reszty dla liczb całkowitych. Powinieneś tak robić w całym kodzie 2.x i 3.x, który piszesz dzisiaj — w pierwszym przypadku dla zachowania kompatybilności z wersją 3.x, a w drugim, ponieważ operator / nie obcina reszty w 3.x. Przykład kodu, na jaki opisana wyżej zmiana sposobu działania operatora / może mieć wpływ, znajdziesz w omówieniu zastosowania pętli while na liczbach pierwszych z rozdziału 13. oraz odpowiadającym mu ćwiczeniu umieszczonem na końcu części czwartej książki. Dodatkowo więcej informacji na temat przedstawionej wyżej specjalnej formy instrukcji import możesz znaleźć w rozdziale 25.

Precyza liczb całkowitych

Choć operacja dzielenia może działać różnie w poszczególnych wersjach Pythona, to jednak jest mechanizmem dosyć standardowym. A zatem czas na coś bardziej egzotycznego. Jak wspominaliśmy już wcześniej, Python 3.x obsługuje liczby całkowite o nieograniczonym rozmiarze:

Python 2.x ma osobny typ `long integer` dla dużych liczb całkowitych i automatycznie przekształca na ten typ wszystkie liczby całkowite, które są zbyt duże, by można je było przechować w zwykłym typie `integer`. Tym samym, aby korzystać z liczb całkowitych typu `long integer`, nie musisz używać żadnej specjalnej składni — jedyną oznaką tego, że w Pythonie 2.x korzystamy z długiej liczby całkowitej, jest wyświetlanie jej ze znakiem `L` na końcu:

Liczby całkowite o nieograniczonej precyzyji są bardzo przydatnym narzędziem wbudowanym. Można ich na przykład użyć przy obliczeniu deficytu narodowego w groszach bezpośrednio w Pythonie (oczywiście jeżeli ktoś ma na to ochotę, a także wystarczającą ilość pamięci w komputerze). Dzięki takim liczbom, w jednym z przykładów z rozdziału 3 mogliśmy podnieść liczbę 2 do bardzo wysokiej potęgi. Oto przykłady działania z wersji 3.x oraz 2.x Pythona:

```
>>> 2 ** 200  
1606938044258990275541962092341162602522202993782792835301376  
>>> 2 ** 200  
1606938044258990275541962092341162602522202993782792835301376L
```

Ponieważ Python musi wykonać dodatkową pracę, by obsługiwać rozszerzoną precyzję takich liczb, arytmetyka liczb całkowitych jest znaczco wolniejsza, kiedy liczby robią się coraz większe. Jeżeli jednak potrzebujemy dodatkowej precyzji, to, że taka możliwość jest od razu wbudowana w język, z pewnością przeważy nad argumentem o jej niższej wydajności.

Liczby zespolone

Choć wykorzystywane są nieco rzadziej niż typy omawiane dotychczas, liczby zespolone są w Pythonie jednym z podstawowych typów obiektów; zazwyczaj mają zastosowanie w aplikacjach inżynierijnych i naukowych. Osoby zatrudnione z nimi z pewnością wiedzą również, dlaczego

liczby te są tak przydatne. Jeżeli nie znasz zastosowań liczb zespolonych, możesz swobodnie potraktować ten fragment tekstu jako opcjonalny.

Liczby zespolone przedstawiane są jako dwie liczby zmiennoprzecinkowe — część rzeczywista i część urojona, które są one oznaczane poprzez dodanie liter `j` lub `J` do części urojonej. Liczby zespolone z niezerową częścią rzeczywistą można również zapisać, łącząc dwie części za pomocą znaku `+`. Poniżej znajdują się proste przykłady obliczeń na liczbach zespolonych:

```
>>> 1j * 1j  
(-1+0j)  
>>> 2 + 1j * 3  
(2+3j)  
>>> (2 + 1j) * 3  
(6+3j)
```

W razie potrzeby możesz również dokonać ekstrakcji części liczb zespolonych jako atrybutów. Liczb zespolonych można używać we wszystkich zwykłych wyrażeniach matematycznych i można je przetwarzać za pomocą narzędzi ze standardowego modułu `cmath` (wersji standardowej modułu `math` przeznaczonej dla liczb zespolonych). Więcej szczegółowych informacji na temat liczb zespolonych można znaleźć w dokumentacji Pythona.

Notacja szesnastkowa, ósemkowa i dwójkowa — literały i konwersje

Liczby całkowite w języku Python mogą być zapisywane w notacji szesnastkowej, ósemkowej i binarnej, oprócz zwykłego zapisu dziesiętnego, którego używaliśmy do tej pory. Pierwsze trzy z nich na pierwszy rzut oka mogą wydawać się nieco obce dla dziesięciopalcowych istot, ale niektórzy programiści uważają je za dogodne alternatywy dla kodowania niektórych wartości, szczególnie gdy ważne jest ich odwzorowanie na bajty i bity. Reguły zapisu w poszczególnych formatach zostały pokrótko przedstawione na początku tego rozdziału, zatem teraz przyszła pora na kilka praktycznych przykładów.

Pamiętaj, że w przykładach przedstawiono tylko alternatywną składnię służącą do określania wartości obiektu liczby całkowitej. Poniższe literały zapisane w Pythonie 3.x oraz 2.x reprezentują zwykłe liczby całkowite zapisane w trzech różnych formatach. Pamiętaj, że w pamięci operacyjnej komputera wartości poszczególnych liczb całkowitych są takie same, niezależnie od podstawy systemu zapisu, którego używamy do ich określania:

```
>>> 0o1, 0o20, 0o377 # Literały ósemkowe: podstawa 8, cyfry  
0-7 (wersja 3.x, 2.6+)  
(1, 16, 255)  
>>> 0x01, 0x10, 0xFF # Literały szesnastkowe: podstawa 16,  
cyfry 0-9/A-F (wersja 3.x, 2.x)  
(1, 16, 255)  
>>> 0b1, 0b10000, 0b11111111 # Literały dwójkowe; podstawa 2, cyfry  
0-1 (3.x, 2.6+)  
(1, 16, 255)
```

Przedstawiona powyżej wartość ósemkowa `0o377`, szesnastkowa `0xFF` i dwójkowa `0b11111111` to dziesiętne 255. Na przykład cyfra `F` w zapisie szesnastkowym odpowiada liczbie 15 w

systemie dziesiętnym i 4-bitowej liczbie 1111 w systemie binarnym. Zatem wartość szesnastkowa 0xFF i inne są konwertowane na wartości dziesiętne w następujący sposób:

```
>>> 0xFF, (15 * (16 ** 1)) + (15 * (16 ** 0))    # Jak liczby hex/bin są
       zamieniane na liczby dziesiętne
(255, 255)
>>> 0x2F, (2 * (16 ** 1)) + (15 * (16 ** 0))
(47, 47)
>>> 0xF, 0b1111, (1*(2**3) + 1*(2**2) + 1*(2**1) + 1*(2**0))
(15, 15, 15)
```

Python domyślnie wyświetla liczby całkowite w formacie dziesiętnym, ale udostępnia wbudowane funkcje pozwalające na konwersję liczb całkowitych na ich odpowiedniki w zapisie o innych podstawach, w formie literałów w języku Python, co jest bardzo przydatne, kiedy program lub użytkownik oczekuje na liczbę zapisaną w danym formacie:

```
>>> oct(64), hex(64), bin(64)                      # Zamiana liczb na ciągi cyfr w
       innych systemach zapisu
('0o100', '0x40', '0b1000000')
```

Funkcja `oct` konwertuje liczbę dziesiętną na ósemkową, funkcja `hex` — na szesnastkową, natomiast `bin` — na dwójkową. Wbudowana funkcja `int` pozwala natomiast przekształcić łańcuch cyfr na liczbę całkowitą, a opcjonalny drugi argument pozwala na określenie podstawy, co jest bardzo użyteczne w przypadku odczytywania z plików liczb kodowanych jako ciągi znaków:

```
>>> 64, 0o100, 0x40, 0b1000000                  # Zamiana ciągów cyfr na liczby
(64, 64, 64, 64)
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)
>>> int('0x40', 16), int('0b1000000', 2)        # Literały też są w porządku
(64, 64)
```

Funkcja `eval`, z którą spotkamy się w dalszej części książki, traktuje łańcuchy znaków tak, jakby były one kodem Pythona. Z tego powodu wykonanie tej funkcji ma podobny efekt jak bezpośrednie wykonanie przekazanego do niej kodu, choć zazwyczaj działa to nieco *wolniej*, gdyż funkcja kompluje oraz wykonuje przekazany do niej ciąg znaków jako część programu i zakłada, że pochodzi on z *zaufanego źródła* — sprytny użytkownik byłby w stanie wysłać odpowiednio spreparowany ciąg znaków, który mógłby na przykład skasować wszystkie pliki z komputera, zatem korzystając z tej funkcji, powinieneś zachować szczególną ostrożność:

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

Warto również wspomnieć, że można przekształcić liczby całkowite na ciągi znaków zapisane w formacie szesnastkowym i ósemkowym za pomocą wywołania metod z *formatującymi ciągami znaków*, które zwracają tylko liczby, a nie literały języka Python:

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64)          #Zamiana cyfr na
       liczby, wersja 2.6+
'100, 40, 1000000'
```

```
>>> '%o, %x, %x, %X' % (64, 64, 255, 255)          # Jak wyżej, we
wszystkich wersjach Pythona
'100, 40, ff, FF'
```

Ciągi formatujące zostały bardziej szczegółowo omówione w rozdziale 7.

Zanim przejdziemy dalej, chciałbym przedstawić dwa ostrzeżenia. Po pierwsze, tak jak wspominaliśmy na początku tego rozdziału, użytkownicy Pythona 2.x powinni pamiętać, że liczby ósemkowe można zapisywać, umieszczając na początku ciągu *wiodące zero*, jak to ma miejsce w oryginalnym formacie ósemkowym Pythona:

```
>>> 0o1, 0o20, 0o377      # Nowy format ósemkowy w wersji 2.6+ (ten sam
co w 3.x)
(1, 16, 255)
>>> 01, 020, 0377        # Stare literały ósemkowe we wszystkich
wersjach 2.x (w 3.x jest to niepoprawny zapis)
(1, 16, 255)
```

W wersji 3.x wyrażenie z drugiego przykładu generuje błąd. Choć w wersji 2.x taki zapis jest najzupełniej poprawny, powinieneś uważać, aby nie rozpoczynać łańcucha cyfr wiodącym zerem, o ile nie chcesz utworzyć wartości ósemkowej. Python 2.x potraktuje taki łańcuch jak liczbę o podstawie 8, co może nie działać tak, jak byś tego oczekwał. W wersji 2.x zapis 010 to zawsze dziesięćne 8, a nie 10 (niezależnie od tego, co o tym myślisz!). Z tego właśnie powodu (a także z uwagi na symetrię zapisu ósemkowego, szesnastkowego i dwójkowego) format ósemkowy został w wersji 3.x zmieniony — w Pythonie 3.x musisz używać zapisu 0o010 i najlepiej zrobić tak samo również w wersjach 2.6 i 2.7, zarówno dla zachowania przejrzystości zapisu, jak i kompatybilności z wersją 3.x.

Po drugie, warto zauważyć, że takie literały mogą tworzyć *dowolnie* długie liczby całkowite. Poniższy kod tworzy na przykład liczbę całkowitą w notacji szesnastkowej, a następnie wyświetla ją najpierw w notacji dziesiętnej, a później ósemkowej oraz dwójkowej, z użyciem konwerterów (w naszym przykładzie kod został uruchomiony w wersji 3.x; w wersji 2.x wartości dziesiętne i ósemkowe mają na końcu dołączoną literę L, wskazującą na typ long integer, a wartości ósemkowe wyświetlane są bez litery o):

```
>>> X = 0xFFFFFFFFFFFFFFFFFFFFFF
>>> X
5192296858534827628530496329220095
>>> oct(X)
'0o17777777777777777777777777777777777777'
>>> bin(X)
'0b111111111111111111111111111111111111111111111 ... i tak dalej...11111'
```

A skoro już mowa o liczbach dwójkowych, w kolejnym podrozdziale zaprezentujemy narzędzia służące do wykonywania operacji na pojedynczych bitach.

Operacje na poziomie bitów

Poza normalnymi operacjami na liczbach (dodawaniem, odejmowaniem i tak dalej) Python obsługuje również większość wyrażeń numerycznych dostępnych w języku C. Obejmuje to także operatory traktujące liczby całkowite jako *ciągi bitów*, które są bardzo użyteczne, kiedy program w języku Python musi sobie radzić z takimi zagadnieniami jak przetwarzanie pakietów

sieciowych, obsługa portów szeregowych czy przetwarzanie spakowanych danych binarnych utworzonych przez program napisany w języku C.

Niestety nie możemy się tutaj bardziej rozwodzić nad podstawami matematyki boolowskiej — podobnie jak poprzednio użytkownicy, którzy muszą z niej korzystać, prawdopodobnie już wiedzą, jak to działa, a inni mogą zwykle odłożyć ten temat na później — ale jej podstawy są proste. Poniżej zamieszczamy kilka przykładów wyrażeń Pythona korzystających z operatorów bitowych, które wykonują operacje przesunięcia bitowego i operacje logiczne na liczbach całkowitych:

```
>>> x = 1                                # Liczba 1 dziesiętnie to 0001 w
      zapisie bitowym

>>> x << 2                                # Przesunięcie o 2 bity w lewo:
0100

4

>>> x | 2                                # Bitowe OR: (dowolny z bitów =
1): 0011

3

>>> x & 1                                # Bitowe AND: (oba bity = 1):
0001

1
```

W pierwszym wyrażeniu binarne 1 (w systemie dwójkowym — 0001) przesuwane jest w lewo o dwa miejsca, tworząc binarne 4 (0100). Ostatnie dwie operacje wykonują binarne OR w celu połączenia bitów (0001 | 0010 = 0011) oraz binarne AND w celu wybrania wspólnych bitów (0001 & 0001 = 0001). Takie operacje na maskach bitowych pozwalają na zakodowanie (lub wyodrębnianie) wielu flag i innych wartości w jednej liczbie całkowitej.

Jest to jeden z obszarów, w których obsługa liczb dwójkowych i szesnastkowych w Pythonie 2.6 oraz 3.0 staje się szczególnie przydatna, ponieważ pozwala na zapisywanie i badanie liczb według ciągów bitów:

```
>>> X = 0b0001                            # Literały dwójkowe

>>> X << 2                                # Przesunięcie w lewo

4

>>> bin(X << 2)                          # Łąncuch cyfr dwójkowych
'0b100'

>>> bin(X | 0b010)                         # Bitowe OR

'0b11'

>>> bin(X & 0b1)                           # Bitowe AND

'0b1'
```

Dotyczy to również wartości, które zaczynają życie jako literały szesnastkowe lub przechodzą konwersję podstawy zapisu:

```
>>> X = 0xFF                            # Literały szesnastkowe

>>> bin(X)

'0b11111111'
```

```

>>> X ^ 0b10101010          # Bitowe XOR (jeden lub drugi,
ale nie oba)

85

>>> bin(X ^ 0b10101010)
'0b1010101'

>>> int('01010101', 2)      # Łańcuch znaków na liczbę
całkowitą zgodnie z podstawą

85

>>> hex(85)                # Liczba na cyfry; ciąg cyfr
szesnastkowych

'0x55'

```

W Pythonie 3.1 i 2.7 wprowadzono dla liczb całkowitych nową metodę `bit_length`, która pozwala na sprawdzenie liczby bitów wymaganej do reprezentacji określonej wartości liczbowej w systemie dwójkowym. Taki sam efekt można osiągnąć, odejmując 2 od długości łańcucha `bin` (aby uwzględnić wiodące „0b”) za pomocą wbudowanej funkcji `len`, którą poznaliśmy w rozdziale 4., chociaż takie rozwiązanie może być nieco mniej wydajne:

```

>>> X = 99
>>> bin(X), X.bit_length(), len(bin(X)) - 2
('0b1100011', 7, 7)
>>> bin(256), (256).bit_length(), len(bin(256)) - 2
('0b100000000', 9, 9)

```

Nie będziemy więcej zajmować się szczegółami opisanych wyżej operacji bitowych. Powinieneś jednak pamiętać, że choć Python obsługuje takie operacje i możesz z nich korzystać, kiedy będzie to niezbędne, to jednak w językach wysokiego poziomu operacje bitowe nie są zwykle tak istotne, jak w językach niskiego poziomu, takich jak C. Zasadniczo, jeżeli chcesz operować na bitach w Pythonie, powinieneś pomyśleć o tym, w jakim języku programujesz. Jak się już się niebawem przekonasz w kolejnych rozdziałach, listy, słowniki i inne tego typu komponenty Pythona dostarczają znacznie bogatszych i zazwyczaj znacznie lepszych sposobów kodowania informacji niż ciągi bitów, szczególnie gdy odbiorcami takich danych są zwyczajni użytkownicy.

Inne wbudowane narzędzia numeryczne

Poza typami podstawowymi Python udostępnia wbudowane *funkcje* i standardowe *moduły bibliotek* służące do przetwarzania danych liczbowych. Na przykład wbudowane funkcje `pow` i `abs` służą do potęgowania i obliczania wartości bezwzględnej liczb. Poniżej zamieszczamy kilka przykładów operacji dostępnych we wbudowanym module `math` (zawierającym większość narzędzi z biblioteki matematycznej języka C) oraz kilka wbudowanych funkcji działających w wersji 3.3+; jak już wspominaliśmy wcześniej, wersje wcześniejsze niż 2.7 i 3.1 mogą wyświetlać liczby zmienoprzecinkowe ze zbyt dużą lub zbyt małą liczbą miejsc po przecinku:

```

>>> import math
>>> math.pi, math.e           # Często używane stałe
(3.141592653589793, 2.718281828459045)
>>> math.sin(2 * math.pi / 180) # Sinus, tangens, cosinus

```

```

0.03489949670250097
>>> math.sqrt(144), math.sqrt(2)          # Pierwiastek kwadratowy
(12.0, 1.4142135623730951)
>>> pow(2, 4), 2 ** 4, 2.0 ** 4.0      # Potęgowanie
(16, 16, 16.0)
>>> abs(-42.0), sum((1, 2, 3, 4))    # Wartość bezwzględna, suma
(42.0, 10)
>>> min(3, 1, 2, 4), max(3, 1, 2, 4)  # Minimum, maksimum
(1, 4)

```

Pokazana w przykładzie funkcja `sum` działa na sekwencji liczb, natomiast `min` i `max` przyjmują albo sekwencję, albo pojedyncze argumenty. Istnieje kilka sposobów na opuszczenie części ułamkowej z liczb zmiennoprzecinkowych. Wcześniej spotkaliśmy się z odcinaniem i zaokrąglaniem do najbliższej mniejszej liczby całkowitej. Możemy również zaokrągać liczby, zarówno liczbowo, jak i na potrzeby wyświetlania:

```

>>> math.floor(2.567), math.floor(-2.567)      # Zaokrąglenie do najbliższej
mniejszej liczby całkowitej
(2, -3)
>>> math.trunc(2.567), math.trunc(-2.567)       # Odcięcie (pominięcie części
ułamkowej)
(2, -2)
>>> int(2.567), int(-2.567)                   # Odcięcie (konwersja na
liczbę całkowitą)
(2, -2)
>>> round(2.567), round(2.467), round(2.567, 2) # Zaokrąglenie (Python 3.x)
(3, 2, 2.57)
>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567)     # Zaokrąglenie na potrzeby
wyświetlania (rozdział 7.)
('2.6', '2.57')

```

Jak widzieliśmy wcześniej, ostatni przykład zwracała łańcuchy znaków, które normalnie byśmy wyświetlili, a także obsługuje różne opcje formatowania. Jak już wspomnieliśmy, w wersjach wcześniejszych niż 2.7 i 3.1 przedostatni przykład zwróciłby (3, 2, 2.57), gdybyśmy umieściły go w wywołaniu funkcji `print`, która wyświetla dane w sposób nieco bardziej przyjazny dla użytkownika. Mimo to te dwa przykłady i tak nieco różnią się od siebie — `round` zaokrąga liczbę zmiennoprzecinkową, jednak przechowuje ją w pamięci, natomiast formatowanie z użyciem łańcucha znaków zwraca ciąg znaków, a nie liczbę:

```

>>> (1 / 3.0), round(1 / 3.0, 2), ('%.2f' % (1 / 3.0))
(0.3333333333333333, 0.33, '0.33')

```

Co ciekawe, w Pythonie istnieją trzy sposoby obliczenia *pierwiastka kwadratowego* — za pomocą funkcji z modułu `math`, za pomocą wyrażenia lub funkcji wbudowanej (osoby zainteresowane wydajnością tych rozwiązań odsyłam do ćwiczeń i ich rozwiązań na końcu czwartej części książki, gdzie sprawdzimy, które z nich działa szybciej).

```

>>> import math
>>> math.sqrt(144)                                # Moduł
12.0
>>> 144 ** .5                                    # Wyrażenie
12.0
>>> pow(144, .5)                                 # Funkcja wbudowana
12.0
>>> math.sqrt(1234567890)                         # Większe liczby
35136.41828644462
>>> 1234567890 ** .5
35136.41828644462
>>> pow(1234567890, .5)
35136.41828644462

```

Warto zauważyć, że moduły biblioteki standardowej, takie jak `math`, trzeba importować, jednak funkcje wbudowane, takie jak `abs` czy `round`, dostępne są zawsze bez importowania. Innymi słowy, moduły to komponenty zewnętrzne, natomiast funkcje wbudowane znajdują się w domyślnej przestrzeni nazw, którą Python automatycznie przeszukuje w celu odnalezienia nazw użytych w programie. Ta przestrzeń nazw odpowiada modułowi o nazwie `builtins` w Pythonie 3.x (`__builtin__` w wersji 2.x). Więcej informacji na temat rozwijania nazw znajdziesz w częściach książki poświęconych funkcjom i modułom, a teraz wystarczy, że kiedy usłyszysz słowo „moduł”, od razu przyjdzie Ci do głowy słowo „importowanie”.

Importować trzeba również moduł biblioteki standardowej `random`, który udostępnia szereg narzędzi służących na przykład do wybierania losowej liczby zmienoprzecinkowej z zakresu od 0 do 1 oraz wybierania losowej liczby całkowitej o wartości z zadanego przedziału:

```

>>> import random
>>> random.random()
0.5566014960423105
>>> random.random()      # Losowe liczby zmienoprzecinkowe, całkowite,
                        wybieranie i tasowanie elementów
0.051308506597373515
>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
9

```

Moduł `random` może również *wybierać* losowo element z sekwencji i losowo przetasować listę przedmiotów:

```

>>> random.choice(['Żywot Briana', 'Święty Graal', 'Sens życia'])
'Swięty Graal'
>>> random.choice(['Żywot Briana', 'Święty Graal', 'Sens życia'])

```

```
'Żywot Briana'  
>>> suits = ['kier', 'karo', 'trefl', 'pik']  
>>> random.shuffle(suits)  
>>> suits  
['pik', 'kier', 'karo', 'trefl']  
>>> random.shuffle(suits)  
>>> suits  
['trefl', 'karo', 'kier', 'pik']
```

Chociaż z pewnością będzie nam potrzebne nieco dodatkowego kodu, aby uczynić to bardziej namacalnym, moduł `random` może być przydatny do tasowania kart w grze, losowego wybierania zdjęć w pokazie slajdów, wykonywania symulacji statystycznych i wielu innych zastosowań. Powróćmy do niego ponownie w dalszej części tej książki (np. w studium przypadku permutacji w rozdziale 20.). Więcej szczegółowych informacji na temat modułu `random` znajdziesz w dokumentacji bibliotek Pythona.

Inne typy liczbowe

W tym rozdziale korzystaliśmy do tej pory wyłącznie z podstawowych typów liczbowych Pythona — liczb całkowitych, liczb zmiennoprzecinkowych oraz liczb zespolonych. Wystarczą one w zupełności do realizacji większości zadań, z jakimi spotka się zazwyczaj przeciętny programista. Python udostępnia jednak bardziej egzotycznych typów liczbowych, które zasługują przynajmniej na krótkie omówienie.

Typ Decimal (liczby dziesiętne)

W wersji 2.4 Pythona wprowadzono nowy podstawowy typ liczbowy — liczbę dziesiętną, formalnie znaną jako `Decimal`. Z punktu widzenia składni liczby te tworzy się, wywołując funkcję z zainportowanego modułu, a nie wyrażenie z literałem. Z funkcjonalnego punktu widzenia liczby dziesiętne przypominają liczby zmiennoprzecinkowe, jednak mają stałą liczbę miejsc dziesiętnych. Tym samym liczba dziesiętna jest wartością zmiennoprzecinkową o stałej precyzyji.

Liczba dziesiętna pozwala na przykład na utworzenie wartości zmiennoprzecinkowej, która zawsze zachowuje dwie pozycje dziesiętne. Co więcej, możemy również określić, w jaki sposób należy zaokrągląć lub odcinać dodatkowe pozycje dziesiętne. Choć liczby dziesiętne są zazwyczaj nieco mniej wydajne od zwykłych liczb zmiennoprzecinkowych, dobrze nadają się do przedstawiania ilości o stałej precyzyji, na przykład pieniędzy, a także mogą pomóc uzyskać lepszą dokładność.

Typ Decimal — zagadnienia podstawowe

Ostatnie stwierdzenie zasługuje na nieco szersze wyjaśnienie. Jak już wspominaliśmy podczas omawiania zagadnień związanych z porównaniami, arytmetyka liczb zmiennoprzecinkowych jest mało dokładna z uwagi na ograniczenie miejsca wykorzystywanego do przechowywania wartości. Na przykład wynikiem działania wyrażenia pokazanego poniżej powinno być zero, jednak tak się nie dzieje. Rezultat jest bardzo zbliżony do zera, jednak nie mamy tutaj wystarczającej liczby bitów do prezentacji dokładnego wyniku.

```
>>> 0.1 + 0.1 + 0.1 - 0.3
```

```
#Python 3.3  
5.551115123125783e-17
```

Wyświetlenie wyniku za pomocą funkcji `print`, w formacie przyjaznym dla użytkownika, niewiele nam tutaj pomoże, ponieważ sprzętowe układy obliczeniowe odpowiedzialne za arytmetykę liczb zmiennoprzecinkowych mają ograniczoną dokładność (co jest związane m.in. z dostępną *precyją* liczb zmiennoprzecinkowych). Wykonanie poniższego polecenia w wersji 3.3 daje taki sam wynik jak poprzednio:

```
>>> print(0.1 + 0.1 + 0.1 - 0.3)      #Wcześniejsze wersje Pythona (w wersji  
3.3 wygląda to inaczej)  
5.55111512313e-17
```

W przypadku liczb dziesiętnych wynik może być dokładny.

```
>>> from decimal import Decimal  
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')  
Decimal('0.0')
```

Jak widać powyżej, obiekty liczb dziesiętnych tworzy się, wywołując funkcję konstruktora `Decimal` znajdująca się w module `decimal` i przekazując jej ciągi znaków o żądanej liczbie miejsc dziesiętnych. Jeżeli jest to niezbędne, możemy wykorzystać funkcję `str` do przekształcenia wartości zmiennoprzecinkowych nałańcuchy znaków. Kiedy w wyrażenach łączymy ze sobą liczby dziesiętne o różnej precyji, Python automatycznie konwertuje je w góre do największej liczby pozycji dziesiętnych.

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')  
Decimal('0.00')
```

W wersjach 2.7, 3.1 i późniejszych możliwe jest również utworzenie obiektu dziesiętnego z obiektu zmiennoprzecinkowego za pomocą wywołania w postaci `decimal.Decimal.from_float(1.25)`, a najnowsze Pythony pozwalają na użycie liczb zmiennoprzecinkowych bezpośrednio. Konwersja jest dokładna, ale czasami może zwracać nadmiernie dużą liczbę cyfr, chyba że ustawiemy to inaczej, tak jak pokażemy to w kolejnym podrozdziale:

```
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)  
Decimal('2.775557561565156540423631668E-17')
```

W Pythonie 3.3 i nowszych moduł `decimal` został również zoptymalizowany w celu poprawienia jego wydajności: deklarowane przez deweloperów przyspieszenie działania nowej wersji wynosi od 10 do 100 razy, w zależności od rodzaju testowanego programu.

Globalne ustawianie precyji liczb dziesiętnych

Inne narzędzia z modułu `decimal` można wykorzystać na przykład do ustawienia precyji wszystkich liczb dziesiętnych, a także aby skonfigurować obsługę błędów. Obiekt kontekstowy z tego modułu pozwala na przykład na określanie precyji (liczby miejsc dziesiętnych) i trybu zaokrąglania (w dół, w górę). Ustawienie precyji stosowane jest globalnie dla wszystkich liczb dziesiętnych utworzonych w wątku wywołującym.

```
>>> import decimal  
>>> decimal.Decimal(1) / decimal.Decimal(7)          # Domyślnie  
28 cyfr  
Decimal('0.1428571428571428571428571429')
```

```

>>> decimal.getcontext().prec = 4 # Stała
precyzja
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3) # Wynik
bliszy zero
Decimal('1.110E-17')

```

Technicznie liczba miejsc znaczących jest określana przez wprowadzane cyfry, a precyza jest stosowana w obliczeniach matematycznych. Jest to szczególnie przydatne w aplikacjach finansowych, w których grosze reprezentowane są jako dwa miejsca ułamkowe. Liczby dziesiętne są właściwie w tym kontekście alternatywą dla ręcznego zaokrąglania i formatowania za pomocą łańcuchów znaków.

```

>>> 1999 + 1.33 # Wynik w pamięci ma większą liczbę miejsc dziesiętnych,
niż jest wyświetlane w wersji 3.3
2000.33
>>>
>>> decimal.getcontext().prec = 2
>>> pay = decimal.Decimal(str(1999 + 1.33))
>>> pay
Decimal('2000.33')

```

Menedżer kontekstu dziesiętnego

W Pythonie 2.6 i 3.0 (oraz kolejnych wersjach) można również tymczasowo przywrócić ustawienia precyzji za pomocą instrukcji `with` udostępnianej przez menedżera kontekstu. Precyza przywracana jest do oryginalnej wartości po wykonaniu polecenia (tak jak wspominaliśmy w rozdziale 3., znaki ... wyświetlane w niektórych interfejsach oznaczają tutaj, że Python oczekuje na kontynuację polecenia w kolejnym wierszu i często wymaga ręcznego ustawienia odpowiedniego wcięcia; w środowisku IDLE taki znak kontynuacji jest pomijany, a edytor środowiska automatycznie ustawia odpowiednie wcięcia):

```

C:\code> C:\Python33\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.333333333333333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
...     ctx.prec = 2
...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')

```

Choć jest to przydatne, instrukcja ta wymaga o wiele większego zasobu wiedzy niż udało nam się do tej pory przedstawić. Więcej szczegółowych informacji na temat instrukcji `with` znajdziesz w rozdziale 34.

Ponieważ korzystanie z typu liczby dziesiętnej nadal jest w praktyce rzadkością, osoby zainteresowane odsyłam do dokumentacji biblioteki standardowej Pythona, a także interaktywnego systemu pomocy. A ponieważ liczby dziesiętne rozwiązuje często te same problemy z dokładnością liczb zmiennoprzecinkowych, co liczby ułamkowe, przejdziemy teraz do kolejnego podrozdziału, aby przekonać się, jak wypada porównanie tych dwóch typów.

Typ Fraction (liczby ułamkowe)

W Pythonie 2.6 oraz 3.0 zadebiutował nowy typ liczbowy — `Fraction` (ułamek), implementujący obiekt *liczby wymiernej*. W sposób jawnym zachowuje on zarówno licznik, jak i mianownik, tak by uniknąć niektórych niedokładności i ograniczeń arytmetyki liczb zmiennoprzecinkowych. Podobnie jak liczby dziesiętne, tak i ułamki nie są odwzorowywane w układach sprzętowych komputera tak dobrze jak liczby zmiennoprzecinkowe. Oznacza to, że w ich przypadku wydajność obliczeń z udziałem ułamków może nie być najlepszą, ale zapewnia dodatkową użyteczność wielu narzędzi.

Typ Fraction – zagadnienia podstawowe

Typ `Fraction` jest funkcjonalnym kuźnem typu dziesiętnego o stałej precyzyji opisanego w poprzedniej sekcji, ponieważ oba mogą być użyte do rozwiązywania niedokładności liczbowych typu zmienoprzecinkowego. Typ ten jest także wykorzystywany w podobny sposób – tak jak `Decimal`, `Fraction` znajduje się w module. Aby utworzyć obiekt tego typu, należy zimportować jego konstruktor i przekazać mu licznik oraz mianownik. Poniższy zapis sesji interaktywnej pokazuje, jak można tego dokonać:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)                                # Licznik, mianownik
>>> y = Fraction(4, 6)                                # Uproszczony do 2, 3 przez gcd
>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

Po utworzeniu obiekty `Fraction` można wykorzystywać w wyrażeniach arytmetycznych w zwykły sposób:

```
>>> x + y
Fraction(1, 1)
>>> x - y
mianownik
# Wyniki sa dokladne: licznik,
Fraction(-1, 3)
>>> x * y
```

```
Fraction(2, 9)
```

Obiekty Fraction mogą również być tworzone z ciągów znaków reprezentujących liczby zmiennoprzecinkowe, podobnie jak liczby dziesiętne:

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)
>>>
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

Dokładność numeryczna ułamków zwykłych i dziesiętnych

Warto zwrócić uwagę na fakt, że dokładność obliczeń na ułamkach różni się od matematyki typu zmiennoprzecinkowego, która jest limitowana przez sprzętowe ograniczenia procesorów numerycznych. Dla porównania poniżej zamieszczamy te same działania wykonane za pomocą obiektów zmiennoprzecinkowych wraz z uwagami dotyczącymi ich ograniczonej dokładności — w nowych wersjach Pythona liczba miejsc po przecinku wyświetlanego na ekranie może być nieco mniejsza niż kiedyś, ale nadal w pamięci nie są to w pełni dokładne wartości:

```
>>> a = 1 / 3.0                      # Dokładność taka, jak procesora do obliczeń
zmiennoprzecinkowych
>>> b = 4 / 6.0                      # Może tracić precyzję w miarę wykonywania
kolejnych obliczeń
>>> a
0.3333333333333333
>>> b
0.6666666666666666
>>> a + b
1.0
>>> a - b
-0.3333333333333333
>>> a * b
0.2222222222222222
```

Ograniczenia liczb zmiennoprzecinkowych są szczególnie widoczne w przypadku wartości, których nie da się reprezentować dokładnie z uwagi na ograniczoną liczbę bitów w pamięci. Zarówno typ Fraction, jak i Decimal zapewniają sposoby uzyskiwania dokładnych wyników obliczeń, aczkolwiek kosztem pewnej szybkości i zwiększenia stopnia złożoności kodu. Na przykład w poniższym przykładzie (powtórzonym z poprzedniego podrozdziału) liczby zmiennoprzecinkowe nie dają oczekiwanej wyniku równego zero, natomiast dwa pozostałe typy robią to bez problemu:

```
>>> 0.1 + 0.1 + 0.1 - 0.3      # To powinno być równe zero (jest blisko, ale
nie dokładnie)
```

```

5.551115123125783e-17
>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')

```

Co więcej, liczby wymierne oraz dziesiętne mogą dawać bardziej intuicyjne i dokładne wyniki, niż czasami robią to liczby zmiennoprzecinkowe, choć w inny sposób — używając reprezentacji wymiernej i ograniczenia precyzji:

```

>>> 1 / 3                                # W Pythonie 2.x należy użyć ".0" dla
uzyskania prawdziwego dzielenia
0.3333333333333333
>>> Fraction(1, 3)                      # Dokładność liczb, dwa sposoby
Fraction(1, 3)
>>> import decimal
>>> decimal.getcontext().prec = 2
>>> Decimal(1) / Decimal(3)
Decimal('0.33')

```

Tak naprawdę liczby ułamkowe zachowują zarówno dokładność, jak i automatycznie upraszczają wyniki. Kontynuując poprzednią sesję interaktywną:

```

>>> (1 / 3) + (6 / 12)                  # W Pythonie 2.x należy użyć ".0" dla
uzyskania prawdziwego dzielenia
0.8333333333333333
>>> Fraction(6, 12)                     # Automatyczne skrócenie ułamka
Fraction(1, 2)
>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)
>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
Decimal('0.83')
>>> 1000.0 / 1234567890
8.100000073710001e-07
>>> Fraction(1000, 1234567890)          # Znacznie prostsze!
Fraction(100, 123456789)

```

Konwersje ułamków i typy mieszane

Aby móc obsługiwać konwersje ułamkowe, obiekty zmiennoprzecinkowe udostępniają teraz metodę zwracającą wartość reprezentującą stosunek licznika do mianownika, ułamki mają

metodę `from_float`, natomiast funkcja `float` przyjmuje obiekt `Fraction` jako argument. Aby przekonać się, jak to działa, wystarczy prześledzić poniższą sesję interaktywną (znak * w drugim teście to składnia specjalna rozszerzająca krotkę na pojedyncze argumenty; więcej informacji na ten temat znajdziesz w omówieniu przekazywania argumentów do funkcji w rozdziale 18.).

```
>>> (2.5).as_integer_ratio()                      # Metoda obiektu liczby
zmienoprzecinkowej
(5, 2)

>>> f = 2.5

>>> z = Fraction(*f.as_integer_ratio()) # Konwersja z liczby
zmienoprzecinkowej na ułamek: 2 argumenty

>>> z                                         # To samo co Fraction(5, 2)
Fraction(5, 2)

>>> x                                         # x z poprzedniej interakcji
Fraction(1, 3)

>>> x + z
Fraction(17, 6)                                # 5/2 + 1/3 = 15/6 + 2/6

>>> float(x)                                    # Konwersja z ułamka na liczbę
zmienoprzecinkową
0.3333333333333333

>>> float(z)
2.5

>>> float(x + z)
2.833333333333335

>>> 17 / 6
2.833333333333335

>>> Fraction.from_float(1.75)                  # Konwersja z liczby
zmienoprzecinkowej na ułamek: inny sposób

Fraction(7, 4)

>>> Fraction(*(1.75).as_integer_ratio())

Fraction(7, 4)
```

Na koniec warto wspomnieć, że mieszanie typów jest w pewnym stopniu dozwolone w wyrażeniach, jednak czasami trzeba ręcznie rozszerzyć typ `Fraction` w celu zachowania dokładności. By przekonać się, jak to działa, warto przyjrzeć się poniższym przykładom.

```
>>> x
Fraction(1, 3)

>>> x + 2                                     # Ułamek + liczba całkowita -> ułamek
Fraction(7, 3)
```

```

>>> x + 2.0                                # Ułamek + liczba zmiennoprzecinkowa ->
liczba zmiennoprzecinkowa
2.3333333333333335

>>> x + (1./3)                            # Ułamek + liczba zmiennoprzecinkowa ->
liczba zmiennoprzecinkowa
0.6666666666666666

>>> x + (4./3)
1.6666666666666665

>>> x + Fraction(4, 3)                   # Ułamek + ułamek -> ułamek
Fraction(5, 3)

```

Uwaga! Choć można dokonać konwersji z liczby zmiennoprzecinkowej na ułamek, w niektórych przypadkach nie do uniknięcia jest pewna utrata precyzji, ponieważ liczba w pierwotnej postaci zmiennoprzecinkowej jest niedokładna. Kiedy jest to niezbędne, można uprościć takie wyniki, ograniczając maksymalną wartość mianownika.

```

>>> 4.0 / 3
1.3333333333333333

>>> (4.0 / 3).as_integer_ratio()          # Utrata precyzji z liczby
zmiennoprzecinkowej
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)

>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)

>>> 22517998136852479 / 13510798882111488. # 5 / 3 (lub wartość zbliżona!)
1.6666666666666667

>>> a.limit_denominator(10)                 # Uproszczenie do najbliższego
ułamka
Fraction(5, 3)

```

Aby uzyskać więcej informacji na temat typu `Fraction`, eksperymentuj dalej samodzielnie i zapoznaj się z podręcznikami bibliotek Python 2.6, 2.7 i 3.X oraz inną dokumentacją.

Zbiory

Oprócz wspomnianego wcześniej typu `Decimal` w Pythonie 2.4 wprowadzony został nowy typ kolekcji — *zbiór* (ang. *set*). Zbiór to nieuporządkowana kolekcja unikalnych i niemutowalnych obiektów, obsługująca działania odpowiadające matematycznej teorii zbiorów. Zgodnie z definicją dany element może się pojawić w zbiorze tylko raz, bez względu na to, ile razy zostanie do niego dodany. Z tego powodu zbiory mają wiele zastosowań, w szczególności w działaniach skupiających się na liczbach oraz bazach danych.

Ponieważ zbiory są kolekcjami innych obiektów, współdzielą z obiektami takimi, jak listy i słowniki pewne zachowania, które pozostają poza zakresem niniejszego rozdziału. Na zbiorach można na przykład wykonywać iterację, mogą one rosnąć i kurczyć się na żądanie, a także zawierać obiekty różnych typów. Jak zobaczymy, zbiór zachowuje się dość podobnie do kluczy słownika bez wartości, jednak obsługuje przy tym dodatkowe operacje.

Ponieważ zbiory są nieuporządkowane i nie odwzorowują kluczy na wartości, nie są ani typem sekwencji, ani odwzorowania — przynależą raczej do własnej kategorii. Co więcej, ponieważ z natury są ściśle związane z matematyką (i wielu Czytelnikom mogą się wydawać bardziej akademickie, a także o wiele rzadziej wykorzystywane niż bardziej powszechnie obiekty, takie jak słowniki), podstawowe zastosowania obiektów zbiorów Pythona omówimy właśnie w tym rozdziale.

Podstawy zbiorów w Pythonie 2.6 i wersjach wcześniejszych

Obecnie istnieje kilka sposobów tworzenia zbiorów — w zależności od wersji Pythona, której używasz. Ponieważ niniejsza książka omawia wszystkie wersje, zaczniemy od przypadku dla wersji 2.6 i wcześniejszych, który jest także dostępny (i czasem nadal wymagany) w późniejszych wersjach; za chwilę poprawimy to dla rozszerzeń 2.7 i 3.x. Aby utworzyć obiekt zbioru, należy przekazać sekwencję lub inny obiekt iterowalny do wbudowanej funkcji `set`.

```
>>> x = set('abcde')
>>> y = set('bxyz')
```

Otrzymujemy w ten sposób obiekt zbioru zawierający wszystkie elementy z przekazanego obiektu (warto zauważyć, że elementy w zbiorach nie są uporządkowane w żadnej określonej kolejności, a więc zbiory nie są sekwencjami — kolejność elementów w zbiorach jest dowolna i może różnić się w zależności od wydania Pythona):

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])                                # Format wyświetlania z wersji
=< 2.6
```

Utworzone w ten sposób zbiory obsługują najważniejsze działania matematyczne na zbiorach za pomocą wyrażeń z operatorami. Warto zauważyć, że nie można takich działań wykonywać na zwykłych sekwencjach, takich jak łańcuchy znaków, listy czy krotki — trzeba z nich najpierw utworzyć zbiory, przekazując je jako argumenty wywołania funkcji `set`:

```
>>> x - y                                              # Różnica zbiorów
set(['a', 'c', 'e'])
>>> x | y                                              # Suma zbiorów
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])
>>> x & y                                              # Część wspólna zbiorów
set(['b', 'd'])
>>> x ^ y                                              # Różnica symetryczna (XOR)
set(['a', 'c', 'e', 'y', 'x', 'z'])
>>> x > y, x < y                                         # Nadzbiór, podzbiór
(False, False)
```

Godnym uwagi wyjątkiem od tej reguły jest wbudowany test przynależności do zbioru, przeprowadzany za pomocą operatora `in` — wyrażenie z tym operatorem może również działać na wszystkich innych typach kolekcji, pozwalając na sprawdzenie, czy dany element należy do kolekcji (jeżeli wolisz myśleć w kategoriach proceduralnych, możesz taką operację nazwać

wyszukiwaniem elementu w zbiorze lub kolekcji), dzięki czemu, chcąc przeprowadzić taki test, nie musisz dokonywać konwersji kolekcji na zbiór:

```
>>> 'e' in x                                # Przynależność (zbiory)
True
>>> 'e' in 'mielonka', 22 in [11, 22, 33]    # Ale działa również na innych
                                               typach
(True, True)
```

Oprócz wyrażeń obiekt zbioru udostępnia również *metody* odpowiadające tym operacjom i nie tylko, obsługujące modyfikacje zbiorów. Metoda `add` wstawia jeden element, `update` to suma zbiorów w miejscu, a `remove` usuwa element o podanej wartości (aby zobaczyć wszystkie dostępne metody, powinieneś wykonać polecenie `dir` na dowolnej instancji zbioru lub nazwie typu zbioru). Zakładając, że `x` i `y` mają wartości jak w poprzedniej sesji interaktywnej:

```
>>> z = x.intersection(y)                      # To samo co x & y
>>> z
set(['b', 'd'])
>>> z.add('MIELONKA')                         # Wstawienie jednego elementu
>>> z
set(['b', 'd', 'MIELONKA'])
>>> z.update(set(['X', 'Y']))                  # Połączenie: suma w miejscu
>>> z
set(['Y', 'X', 'b', 'd', 'MIELONKA'])
>>> z.remove('b')                             # Usunięcie jednego elementu
>>> z
set(['Y', 'X', 'd', 'MIELONKA'])
```

Ponieważ zbiory są kontenerami *iterowalnymi*, można je także wykorzystywać w operacjach takich jak `len`, pętle `for` czy listy składane. Warto jednak pamiętać, że ponieważ zbiory nie są uporządkowane, nie obsługują operacji na sekwencjach, takich jak indeksowanie czy wycinki.

```
>>> for item in set('abc'): print(item * 3)
aaa
ccc
bbb
```

Wreszcie, choć zaprezentowane powyżej wyrażenia zbiorów zazwyczaj wymagają podania dwóch zbiorów, odpowiadające im metody działają także na *dowolnych innych typach iterowalnych*:

```
>>> S = set([1, 2, 3])
>>> S | set([3, 4])                           # Wyrażenie wymaga, by oba
                                               operandy były zbiorami
set([1, 2, 3, 4])
>>> S | [3, 4]
```

```

TypeError: unsupported operand type(s) for |: 'set' and 'list'
>>> S.union([3, 4])                                # Metody zbiorów akceptują
dowolne typy iterowalne

set([1, 2, 3, 4])

>>> S.intersection([1, 3, 5])

set([1, 3])

>>> S.issubset(range(-5, 5))

True

```

Więcej informacji na temat działań na zbiorach można znaleźć w dokumentacji biblioteki Pythona lub książkach. Choć działania na zbiorach można w Pythonie kodować ręcznie za pomocą innych typów, takich jak listy i słowniki (i często tak właśnie kiedyś się robiło), wbudowane zbiory Pythona wykorzystują wydajne algorytmy i techniki implementacyjne, które ułatwiają szybkie wykonywanie standardowych operacji.

Literały zbiorów w Pythonie 3.x i 2.7

Jeżeli uważasz, że zbiory są „fajne”, to z pewnością ucieszysz się, że w nowych wersjach Pythona stały się jeszcze fajniejsze, z nową składnią literałów i wyrażeń, które początkowo były dostępne tylko w linii Pythona 3.x, ale ze względu na dużą popularność i naciski ze strony użytkowników zostały również zaimplementowane w wersji 2.7. W tych wersjach nadal możemy tworzyć obiekty zbiorów za pomocą wbudowanego konstruktora `set`, jednak dodano również nową formę literala zbiorów, wykorzystującą nawiasy klamrowe zarezerwowane wcześniej dla słowników. W wersjach 3.x i 2.7 następujące wyrażenia są równoważne:

```

set([1, 2, 3, 4])                                # Wywołanie wbudowanego
konstruktora (wszystkie wersje)

{1, 2, 3, 4}                                     # Nowe literały zbiorów (wersje
2.7, 3.x)

```

Powyższa składnia ma sens, biorąc pod uwagę to, że zbiory są właściwie jak *słowniki bez wartości* — ponieważ elementy zbioru są nieuporządkowane, unikalne i niemutowalne, zachowują się jak klucze w słownikach. To podobieństwo w działaniu staje się jeszcze bardziej uderzające, gdy weźmiemy pod uwagę, że w wersji 3.x listy kluczy słowników są obiektami *widoku*, obsługującymi działania podobne do zbiorów, takie jak część wspólna czy suma (więcej informacji o obiektach widoku słowników znajdziesz w rozdziale 8.).

Niezależnie od tego, jak tworzone są zbiory, Python 3.x wyświetla je za pomocą nowego formatu literałów. Python 2.7 akceptuje nową składnię literałów, ale nadal wyświetla zbiory tak jak wersja 2.6, co pokazywaliśmy w poprzednim podrozdziale. We wszystkich wersjach Pythona wbudowany konstruktor `set` jest nadal niezbędny do tworzenia pustych zbiorów i budowania zbiorów z istniejących obiektów iterowalnych (oprócz wykorzystywania zbiorów składanych omówionych w dalszej części rozdziału), ale nowy literał przydaje się do inicjowania zbiorów o znanej strukturze.

Zbiory w wersji 2.7 wyglądają tak samo jak w wersji 3.x, tyle że zawartość zbiorów wyświetlana jest w przyjętej w wersji 2.x notacji `set([...])`, a kolejność elementów może się różnić w zależności od wersji (a z zasady kolejność elementów w zbiorach i tak nie ma żadnego znaczenia):

```

C:\code> c:\python33\python

>>> set([1, 2, 3, 4])                            # Wbudowany konstruktor: tak samo jak w
2.6

{1, 2, 3, 4}

```

```

>>> set('mielonka')                                # Dodanie wszystkich elementów obiektu
iterowalnego
{'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n'}
>>> {1, 2, 3, 4}                                    # Literały zbiorów: nowość w 3.x i 2.7
{1, 2, 3, 4}
>>> S = {'m', 'i', 'e', 'l', 'o', 'n', 'k', 'a'}
>>> S
{'n', 'o', 'k', 'm', 'a', 'l', 'i', 'e'}
>>> S.add('konserwowa')                           #Metoda add działa jak poprzednio
>>> S
{'n', 'o', 'k', 'm', 'a', 'l', 'i', 'konserwowa', 'e'}

```

Wszystkie omówione w poprzednim podrozdziale operacje przetwarzające zbiory działają w Pythonie 3.x tak samo, lecz zbiory wynikowe wyświetlane są w nieco odmienny sposób:

```

>>> S1 = {1, 2, 3, 4}
>>> S1 & {1, 3}                                     # Część wspólna
{1, 3}
>>> {1, 5, 3, 6} | S1                             # Suma
{1, 2, 3, 4, 5, 6}
>>> S1 - {1, 3, 4}                               # Różnica
{2}
>>> S1 > {1, 3}                                   # Nadzbiór
True

```

Warto pamiętać, że {} jest w Pythonie nadal słownikiem. *Puste* zbiory muszą być tworzone za pomocą wbudowanego konstruktora set i w ten sam sposób wyświetlane.

```

>>> S1 - {1, 2, 3, 4}                            # Puste zbiory wyświetlane są
inaczej
set()
>>> type({})                                     # Ponieważ {} jest pustym
słownikiem
<class 'dict'>
>>> S = set()                                    # Inicjacja pustego zbioru
>>> S.add(1.23)
>>> S
{1.23}

```

Podobnie jak w Pythonie 2.6 i wersjach wcześniejszych, zbiory utworzone za pomocą literałów z wersji 3.x/2.7 obsługują te same metody, a niektóre z nich pozwalają nawet na wykorzystywanie operandów iterowalnych, których nie można używać w wyrażeniach.

```
>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'
>>> {1, 2, 3}.union([3, 4])
{1, 2, 3, 4}
>>> {1, 2, 3}.union({3, 4})
{1, 2, 3, 4}
>>> {1, 2, 3}.union(set([3, 4]))
{1, 2, 3, 4}
>>> {1, 2, 3}.intersection((1, 3, 5))
{1, 3}
>>> {1, 2, 3}.issubset(range(-5, 5))
True
```

Ograniczenia na obiekty niemutowalne i zbiory zamrożone

Zbiory są elastycznymi obiektami o dużych możliwościach, jednak w Pythonie 2.x i 3.x mają jedno ograniczenie, o którym należy pamiętać — ze względu na sposób implementacji zbiory mogą zawierać jedynie obiekty *niemutowalne*. Tym samym w zbiorach nie można osadzać list i słowników, choć można to zrobić w przypadku krotek, jeżeli niezbędne jest przechowanie wartości złożonych. Przy wykorzystywaniu w działańach na zbiorach krotki porównywane są pod względem pełnej wartości.

```
>>> S
{1.23}
>>> S.add([1, 2, 3])                                # W zbiorze mogą znajdować się
jedynie obiekty niemutowalne
TypeError: unhashable type: 'list'
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'
>>> S.add((1, 2, 3))
# Ani listy, ani słowniki; krotki
są OK
{1.23, (1, 2, 3)}
>>> S | {(4, 5, 6), (1, 2, 3)}                  # Suma: to samo co S.union(...)
{1.23, (4, 5, 6), (1, 2, 3)}
>>> (1, 2, 3) in S                               # Istnienie w zbiorze: porównanie
po pełnej wartości
True
>>> (1, 4, 3) in S
```

`False`

Krotki można w zbiorach wykorzystać do reprezentowania dat, rekordów, adresów IP i tak dalej (więcej informacji o krotkach znajdziesz w dalszej części książki). Same zbiory są mutowalne, dlatego nie mogą być zagnieżdżane w innych zbiorach w sposób bezpośredni; jeżeli jednak przechowanie zbioru wewnętrz innego zbioru jest konieczne, wywołanie wbudowanego konstruktora `frozenset` działa tak samo jak `set`, ale tworzy zbiór niemutowalny, którego nie można modyfikować, dzięki czemu można go osadzać w innych zbiorach.

Zbiory składane w Pythonie 3.x i 2.7

Obok literałów w Pythonie 3.x wprowadzono także konstrukcję zbioru składanego (ang. *set comprehension*), która została później zaimplementowana również w Pythonie 2.7. Podobnie jak to było w przypadku literałów zbiorów z wersji 3.x, Python 2.7 akceptuje składnię zbiorów składanych, ale wyświetla rezultaty w notacji przyjętej w linii 2.x. Jest ona podobna do list składanych omówionych pokrótko w rozdziale 4., jednak zapisywana jest w nawiasach klamrowych zamiast kwadratowych i wykonywana w celu utworzenia zbioru, a nie listy. Zbiory składane przetwarzane są w pętli i zbierają wynik wyrażenia z każdą iteracją. Zmienna pętli daje dostęp do aktualnej wartości iteracji, z której można skorzystać w wyrażeniu zbierającym. Wynikiem jest nowy zbiór utworzony w rezultacie wykonania kodu, posiadający normalne właściwości zbiorów. Poniżej przedstawiamy przykład zbioru składanego w wersji 3.3 (pamiętaj, że sposób wyświetlania i kolejność elementów w wersji 2.7 może być inna):

```
>>> {x ** 2 for x in [1, 2, 3, 4]}          # Zbiory składane z wersji  
3.x/2.7  
{16, 1, 4, 9}
```

W powyższym wyrażeniu pętla zapisywana jest po prawej stronie, natomiast wyrażenie zbierające po lewej (`x ** 2`). Tak jak w przypadku list składanych, otrzymujemy właściwie to, co mówi wyrażenie: „Utwórz nowy zbiór zawierający `x` podniesione do kwadratu dla każdego `x` z listy”. Zbiory składane mogą również wykonywać iterację po innych typach obiektów, takich jak łańcuchy znaków (pierwszy z poniższych przykładów ilustruje oparty na zbiorach składanych sposób utworzenia zbioru z istniejącego elementu iterowalnego).

```
>>> {x for x in 'mielonka'}                  # To samo co: set('mielonka')  
{'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n'}  
>>> {c * 4 for c in 'mielonka'}            # Zbiór zebranych wyników  
wyrażenia  
{'iiii', 'eeee', 'oooo', 'nnnn', 'mmmm', 'aaaa', 'llll', 'kkkk'}  
>>> {c * 4 for c in 'szynkajajka'}  
{'yyyy', 'jjjj', 'nnnn', 'zzzz', 'ssss', 'aaaa', 'kkkk'}  
>>> S = {c * 4 for c in 'mielonka'}  
>>> S | {'mmmm', 'xxxx'}  
{'mmmm', 'eeee', 'xxxx', 'oooo', 'nnnn', 'iiii', 'aaaa', 'llll', 'kkkk'}  
>>> S & {'mmmm', 'xxxx'}  
{'mmmm'}
```

Ponieważ inne właściwości zbiorów składanych oparte są na koncepcjach, których jeszcze nie omawialiśmy, odłożymy ich przedstawienie do późniejszej części książki. W rozdziale 8. spotkamy pierwszych krewnych zbiorów składanych z wersji 3.x i 2.7 — słowniki składane. O wiele więcej na temat wszystkich typów obiektów składanych (list, zbiorów, słowników i generatorów) będę miał do powiedzenia później, zwłaszcza w rozdziałach 14. oraz 20. Jak się

przekonamy, wszystkie obiekty składane, w tym zbiory, obsługują dodatkową składnię wykraczającą poza pokazaną tutaj, w tym zagnieździone pętle i testy `if`, co może być trudne do zrozumienia, dopóki nie omówimy bardziej złożonych instrukcji.

Dlaczego zbiory?

Działania na zbiorach mają wiele powszechnie wykorzystywanych zastosowań; niektóre z nich są bardziej praktyczne niż matematyczne. Przykładowo, ponieważ elementy przechowywane są w zbiorze są unikatowe, zbiory można wykorzystywać do *odfiltrowywania duplikatów* z kolekcji innego typu (choć należy pamiętać, że w wyniku takiej operacji kolejność elementów może się zmienić, ponieważ zbiory z definicji nie są uporządkowane). Aby to zrobić, wystarczy przekształcić kolekcję na zbiór, a następnie dokonać przekształcenia otrzymanego zbioru z powrotem na kolekcję (zbiory są *iterowalne*, dlatego możemy ich użyć jako argumentu wywołania funkcji `list`; jest to kolejna techniczna ciekawostka, o której będziemy jeszcze pisać nieco później):

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                                     # Usuwamy duplikaty
>>> L
[1, 2, 3, 4, 5]
>>> list(set(['yy', 'cc', 'aa', 'xx', 'dd', 'aa'])) # ...ale kolejność
elementów może ulec zmianie
['cc', 'xx', 'yy', 'dd', 'aa']
```

Zbiory mogą być również używane do *izolowania różnic* w listach, ciągach znaków i innych obiektach iterowalnych — w tym celu wystarczy dokonać konwersji takiego elementu na zbiór i zapisać różnicę — choć w takiej sytuacji nieuporządkowany charakter zbiorów oznacza, że otrzymane wyniki mogą nie pasować do oryginału. Dwa ostatnie przykłady poniżej porównują atrybuty typów obiektów łańcuchowych w wersji 3.x (w wersji 2.7 wyniki mogą być inne):

```
>>> set([1, 3, 5, 7]) - set([1, 2, 4, 5, 6])      # Wyszukiwanie różnic w
listach
{3, 7}
>>> set('abcdefg') - set('abdghij')                 # Wyszukiwanie różnic w
ciągach znaków
{'c', 'e', 'f'}
>>> set('spam') - set(['h', 'a', 'm'])              # Wyszukiwanie różnic, tryb
mieszany
{'p', 's'}
>>> set(dir(bytes)) - set(dir(bytarray))           # Różnica atrybutów między
typem bytes a bytarray
{'__getnewargs__'}
>>> set(dir(bytarray)) - set(dir(bytes))
{'append', 'copy', '__alloc__', '__imul__', 'remove', 'pop', 'insert', ...i
tak dalej...]
```

Możesz także używać zbiorów do przeprowadzania *testów równości niezależnych od kolejności elementów*, konwertując obiekt iterowalny na zbiór przed wykonaniem testu, ponieważ jak już wspominaliśmy, kolejność elementów w zbiorach nie ma znaczenia. Mówiąc bardziej formalnie, dwa zbiorów są równe wtedy i tylko wtedy, gdy każdy element każdego zbioru jest zawarty w drugim zbiorze — to znaczy każdy ze zbiorów jest podzbiorem tego drugiego, niezależnie od kolejności. Możesz na przykład użyć tego do porównania wyników działania programów, które powinny działać tak samo, ale mogą generować wyniki w różnej kolejności. Sortowanie przed testowaniem również pozwala na przeprowadzanie testów równości, ale zbiorów nie opierają się na kosztownym sortowaniu; z kolei posortowane elementy wspierają dodatkowe testy wielkości, których zbiorów nie obsługują (np. większe niż, mniejsze niż itd.):

```
>>> L1, L2 = [1, 3, 5, 2, 4], [2, 5, 3, 4, 1]
>>> L1 == L2                                     # Kolejność elementów w
sekwencjach ma znaczenie
False
>>> set(L1) == set(L2)                         # Testowanie równości niezależne
od kolejności elementów
True
>>> sorted(L1) == sorted(L2)                   # Podobne działanie, ale wyniki
są posortowane
True
>>> 'spam' == 'asmp', set('spam') == set('asmp'), sorted('spam') ==
sorted('asmp')
(False, True, True)
```

Zbiorów można także wykorzystać do śledzenia miejsc, w których już byłeś podczas poruszania się po grafie czy innej *strukturze cyklicznej*. W przykładach dotyczących przeładowywania modułów czy wyświetlania drzewa dziedziczenia, o których będziemy mówić w rozdziałach 24. i 31., będziemy śledzić odwiedzane elementy, aby uniknąć zapętlenia (w rozdziale 19. znajdziesz krótkie streszczenie tego zagadnienia). Używanie listy w takim kontekście jest bardzo nieefektywne, ponieważ wyszukiwanie wymaga skanowania liniowego. Choć zapisywanie stanów odwiedzonych w postaci kluczy słownika jest dosyć wydajne, zastosowanie zbiorów jest alternatywą, która jest właściwie równoważna (i bardziej lub mniej intuicyjna w zależności od tego, kogo o to zapytamy).

Wreszcie zbiorów przydają się w przypadku konieczności przetwarzania większych zestawów danych (na przykład wyników zapytań do bazy danych). Część wspólna dwóch zbiorów zawiera obiekty powtarzające się w obu, natomiast suma zbiorów to wszystkie elementy znajdujące się w którymkolwiek z nich. Poniżej znajduje się nieco bardziej realistyczny przykład działań na zbiorach, zastosowany do list pracowników hipotetycznej firmy i wykorzystujący literał zbiorów z wersji 3.x/2.7 oraz wyświetlanie wyników z wersji 3.x (w wersji 2.6 i wcześniejszych należy użyć konstruktora set).

```
>>> engineers = {'robert', 'amadeusz', 'anna', 'aleksander'}
>>> managers = {'edward', 'amadeusz'}
>>> 'robert' in engineers                      # Czy Robert jest inżynierem?
True
>>> engineers & managers                      # Kto jest inżynierem i
menedżerem?
{'amadeusz'}
```

```

>>> engineers | managers          # Wszystkie osoby z dowolnej
kategorii
{'edward', 'amadeusz', 'anna', 'robert', 'aleksander'}
>>> engineers - managers        # Inżynierowie, którzy nie są
menedżerami
{'robert', 'aleksander', 'anna'}
>>> managers - engineers        # Menedżerowie, którzy nie są
inżynierami
{'edward'}
>>> engineers > managers        # Czy wszyscy menedżerowie są
inżynierami? (nadzbiór)
False
>>> {'robert', 'amadeusz'} < engineers      # Czy obaj są inżynierami?
(podzbiór)
True
>>> (managers | engineers) > managers      # Nadzbiorem menedżerów są
wszyscy pracownicy
True
>>> managers ^ engineers         # Kto jest w jednym zbiorze, ale
nie obu?
{'robert', 'edward', 'aleksander', 'anna'}
>>> (managers | engineers) - (managers ^ engineers) # Część wspólna!
{'amadeusz'}

```

Więcej informacji na temat działań na zbiorach można znaleźć w dokumentacji biblioteki Pythona, a także tekstach teoretycznych poświęconych matematyce i relacyjnym bazom danych. W rozdziale 8. powróćmy do niektórych omówionych tutaj operacji na zbiorach w kontekście obiektów widoku słowników z Pythona 3.x.

Wartości Boolean

Niektóre osoby uważają, że typ Boolean w języku Python (`bool`) jest z natury liczbowy, ponieważ jego dwie wartości — `True` i `False` — są tylko innymi wersjami liczb całkowitych `1` i `0`, które jedynie zapisuje się w odmienny sposób. Choć większość programistów taka wiedza wystarczy, omówimy ten typ nieco bardziej szczegółowo.

Formalnie Python ma jawnego typ danych Boolean o nazwie `bool`, z wartościami `True` i `False` dostępnymi jako wbudowane nazwy. Wewnętrznie te nazwy są instancjami klasy `bool`; `bool` z kolei jest tylko podklassą (w znaczeniu zorientowania obiektowego) wbudowanego typu liczby całkowitej `int`. `True` i `False` zachowują się dokładnie tak samo jak liczby całkowite `1` i `0`, jednak mają własną logikę wyświetlania — wyświetlane są jako słowa `True` i `False` w miejsce cyfr `1` i `0`. Tak naprawdę `bool` redefiniuje formaty `str` i `repr` na potrzeby swoich dwóch obiektów.

Ze względu na tę zmianę wyniki działania wyrażeń Boolean wpisywanych w sesji interaktywnej wyświetlane są jako słowa `True` i `False` zamiast starszej, mniej oczywistej wersji `1` i `0`, a dodatkowo powoduje to, że wartości typu Boolean są w kodzie programu bardziej wyeksponowane. Na przykład pętlę można teraz zapisać jako `while True:` zamiast mniej

intuicyjnego `while 1:`. W podobny sposób można także inicjować flagi za pomocą wyrażeń takich jak `flag = False`. Więcej szczegółowych informacji na temat takich instrukcji znajdziesz jeszcze w trzeciej części książki.

Dla wszystkich innych celów praktycznych można traktować `True` i `False` tak, jakby były zdefiniowanymi zmiennymi o wartości 1 i 0. Większość programistów i tak przypiszywała `True` i `False` do liczb 1 i 0, więc nowy typ jedynie standaryzuje tę technikę. Jej implementacja może jednak w pewnych sytuacjach prowadzić do dziwnych wyników — ponieważ `True` jest po prostu liczbą całkowitą 1 z innym formatem wyświetlania, `True + 4` zwraca w Pythonie wynik 5!

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> True == 1                                # Ta sama wartość
True
>>> True is 1                               # Ale inny obiekt; patrz
      kolejny rozdział
False
>>> True or False                           # To samo co: 1 or 0
True
>>> True + 4                                # (Hmmm)
5
```

Ponieważ prawdopodobnie nigdy nie spotkasz takiego wyrażenia jak to ostatnie w kodzie prawdziwego programu w języku Python, możesz bezpiecznie zignorować jego głębokie implikacje metafizyczne...

Wartościami Boolean zajmiemy się ponownie w rozdziale 9. (przy okazji definiowania pojęcia prawdy w Pythonie), a także w rozdziale 12. (aby sprawdzić, jak działają operatory logiczne takie jak `and` czy `or`).

Rozszerzenia numeryczne

Oprócz własnych typów liczbowych Pythona istnieje również wiele różnych dodatków na licencji open source, przeznaczonych do bardziej wyspecjalizowanych zastosowań. Ponieważ programowanie numeryczne jest popularną dziedziną zastosowania Pythona, dostępnych jest wiele różnych zaawansowanych narzędzi.

Na przykład, jeżeli chcesz wykonywać jakieś poważniejsze obliczenia, opcjonalne rozszerzenie Pythona o nazwie `NumPy` (Numeric Python) udostępnia zaawansowane narzędzia programowania numerycznego, takie jak macierze, przetwarzanie wektorów i rozbudowane biblioteki do zaawansowanych obliczeń numerycznych. Poważne zespoły naukowców w miejscowościach takich jak Los Alamos czy NASA wykorzystują Pythona w połączeniu z NumPy do implementacji zadań, które wcześniej wykonywali w językach C++, FORTRAN czy programie Matlab. Połączenie Pythona i NumPy jest często określane mianem darmowej i bardziej elastycznej wersji Matlaba — otrzymujemy wydajność pakietu NumPy wraz z pełną elastycznością i funkcjonalnością Pythona z jego bibliotekami.

Ponieważ pakiet NumPy jest bardzo specjalistycznym rozszerzeniem Pythona, nie będziemy go omawiać w niniejszej książce. Dodatkowe wsparcie dla zaawansowanego programowania numerycznego w Pythonie, w tym narzędzia do tworzenia wykresów, liczby zmiennoprzecinkowe o rozszerzonej precyzyji, biblioteki statystyczne, a także popularny pakiet *SciPy*, można bez trudu znaleźć, przeszukując internet. Warto również podkreślić, że NumPy jest obecnie rozszerzeniem opcjonalnym; nie jest częścią Pythona i musi być zainstalowany osobno. Nie jest to jednak trudne i jeżeli będzie Ci na tym zależało, w internecie znajdziesz szczegółowe wskazówki, jak to zrobić.

Podsumowanie rozdziału

Niniejszy rozdział zawierał przegląd typów obiektów liczbowych Pythona, a także operacji, jakie można do nich zastosować. Poznaliśmy standardowe typy liczb całkowitych i zmiennoprzecinkowych, a także typy bardziej egzotyczne i rzadziej używane, jak liczby zespolone, liczby dziesiętne, ułamki i zbiory. Omówiliśmy również składnię wyrażeń Pythona, konwersję typów, operacje poziomu bitowego i różne literały służące do kodowania liczb w programach.

W dalszej części książki uzupełnimy szczegółowe dotyczące kolejnego typu danych, czyli łańcucha znaków. W kolejnym rozdziale poświęcimy jednak sporo czasu szczegółowemu omówieniu mechanizmu przypisywania zmiennych. Jest to jedna z najbardziej podstawowych koncepcji w języku Python, więc zanim przejdziesz dalej, powinieneś się zapoznać z tym rozdziałem. Najpierw jednak pora na zwyczajowy quiz.

Sprawdź swoją wiedzę — quiz

1. Jaka jest wartość wyrażenia $2 * (3 + 4)$ w Pythonie?
2. Jaka jest wartość wyrażenia $2 * 3 + 4$ w Pythonie?
3. Jaka jest wartość wyrażenia $2 + 3 * 4$ w Pythonie?
4. Jakich narzędzi można użyć do obliczenia pierwiastka kwadratowego danej liczby, a także jej kwadratu?
5. Jakiego typu będzie wynik wyrażenia $1 + 2.0 + 3$?
6. W jaki sposób można odciąć i zaokrąglić liczbę zmiennoprzecinkową?
7. W jaki sposób można przekonwertować liczbę całkowitą na liczbę zmiennoprzecinkową?
8. W jaki sposób można wyświetlić liczbę całkowitą w notacji szesnastkowej, ósemkowej czy dwójkowej?
9. W jaki sposób można przekonwertować łańcuch ósemkowy, szesnastkowy lub dwójkowy na zwykłą liczbę całkowitą?

Sprawdź swoją wiedzę — odpowiedzi

1. Wynikiem będzie 14, czyli $2 * 7$, gdyż nawiasy wymuszają wykonanie dodawania przed mnożeniem.
2. Teraz wynikiem będzie 10, czyli $6 + 4$. Gdy nie ma nawiasów, stosowane są reguły priorytetu operatorów Pythona, a mnożenie ma wyższy priorytet od dodawania (czyli wykonywane jest wcześniej) — zgodnie z tabelą 5.2.
3. Wyrażenie to zwraca 14, czyli $2 + 12$, zgodnie z tymi samymi regułami priorytetu co w poprzednim pytaniu.
4. Funkcje służące do uzyskania pierwiastka kwadratowego, a także na przykład liczby pi i czy tangensa, można znaleźć w importowanym module `math`. Aby obliczyć pierwiastek kwadratowy liczby, należy zaimportować moduł `math` i wywołać funkcję `math.sqrt(N)`. Aby otrzymać kwadrat liczby, należy albo wykorzystać wyrażenie potęgowania $X ** 2$, albo wbudowaną funkcję `pow(X, 2)`. Oba ostatnie rozwiązania mogą także obliczyć pierwiastek kwadratowy po podaniu potęgi 0.5 (na przykład $X ** .5$).
5. Wynik będzie liczbą zmiennoprzecinkową — liczby całkowite przekształcane są w górę do liczb zmiennoprzecinkowych, najbardziej skomplikowanego typu tego wyrażenia. Do obliczenia wykorzystana zostanie arytmetyka liczb zmiennoprzecinkowych.
6. Funkcje `int(N)` oraz `math.trunc(N)` odcinają część ułamkową, natomiast `round(N, liczba_cyfr)` zaokrąglą ją. Możemy także obliczyć zaokrąglenie do najbliższej mniejszej liczby całkowitej za pomocą `math.floor(N)` i zaokrąglić liczbę do wyświetlenia za pomocą operacji formatowania łańcucha znaków.
7. Funkcja `float(I)` konwertuje liczbę całkowitą na zmiennoprzecinkową. Podobna konwersja będzie wynikiem połączenia w jednym wyrażeniu liczb obu tych typów. W pewnym sensie dzielenie `/` z Pythona 3.x także wykonuje konwersję — zwraca liczbę zmiennoprzecinkową obejmującą resztę, nawet jeśli oba argumenty były liczbami całkowitymi.
8. Wbudowane funkcje `oct(I)` i `hex(I)` zwracają łańcuch ósemkowy i szesnastkowy dla podanej liczby całkowitej. Wbudowana funkcja `bin(I)` zwraca z kolei łańcuch cyfr dwójkowych w Pythonie 2.6 oraz 3.0 i nowszych. To samo można osiągnąć za pomocą wyrażenia z formatującym łańcuchem znaków `%` oraz metody łańcuchów znaków `format`.
9. Funkcję `int(S, podstawa)` można wykorzystać do przekształcenia łańcucha ósemkowego lub szesnastkowego na normalną liczbę całkowitą (jako podstawę należy podać 8, 16 lub 2). Do tego samego celu można również wykorzystać funkcję `eval(S)`, jednak jej wykonanie jest bardziej kosztowne i może się wiązać z problemami z bezpieczeństwem. Warto zauważyć, że liczby całkowite są w pamięci komputera zawsze przechowywane w postaci binarnej; pozostałe formy to tylko konwersje formatu wyświetlania.

Rozdział 6. Wprowadzenie do typów dynamicznych

W poprzednim rozdziale rozpoczęliśmy szczegółowe omawianie podstawowych typów obiektów Pythona, zaczynając od liczb. Wróćmy do przewodnika po typach obiektów w kolejnym rozdziale, ale zanim przejdziemy dalej, ważne jest, abyś zapoznał się z czymś, co może być najbardziej fundamentalną koncepcją programowania w Pythonie, a z pewnością jest podstawą dużej zwięzłości i elastyczności tego języka — typy dynamiczne i związane z nimi polimorfizm.

Jak zobaczymy zarówno tutaj, jak i w dalszej części książki, w Pythonie nie deklarujemy typu obiektów wykorzystywanych przez skrypty. W rzeczywistości większość programów nie powinna nawet *dbać* o określone typy; w zamian za to mogą one być przydatne w większej liczbie zastosowań, niż czasami sami planujemy. Ponieważ podstawą tej elastyczności są typy dynamiczne, które jednocześnie mogą być potencjalną przeszkodą dla początkujących użytkowników Pythona, w tym rozdziale spróbujmy się przyjrzeć im nieco bliżej.

Sprawa brakujących deklaracji typu

Jeżeli masz doświadczenie z komplikowanymi lub statycznie wpisywanymi językami, takimi jak C, C++ lub Java, możesz się tutaj czuć nieco zakłopotany. Dotychczas wykorzystywaliśmy zmienne bez deklarowania ich istnienia czy typów i jakoś to działało. Kiedy w sesji interaktywnej czy pliku programu wpiszemy na przykład `a = 3`, w jaki sposób Python będzie wiedział, że `a` to liczba całkowita? A skoro już przy tym jesteśmy, skąd Python w ogóle wie, czym jest `a`?

Kiedy zaczynamy sobie zadawać takie pytania, wkraczamy do modelu *typów dynamicznych* Pythona. W Pythonie typy ustalane są automatycznie w momencie wykonywania, a nie w odpowiedzi na deklarację w kodzie. Oznacza to, że nigdy nie deklarujemy zmiennych wcześniej (koncepcję tę łatwiej można pojąć, kiedy się pamięta, że wszystko sprowadza się do zmiennych, obiektów i połączeń między nimi).

Zmienne, obiekty i referencje

Jak widzieliśmy w wielu przykładach przedstawionych w tej książce, kiedy w Pythonie wykonywana jest instrukcja przypisania, na przykład `a = 3`, działa ona mimo tego, że nigdy Pythonowi nie nakazaliśmy użyć `a` jako zmiennej ani też nie określiliśmy, że `a` ma być obiektem liczby całkowitej. W tym języku wszystko to układają się w sposób bardzo naturalny.

Tworzenie zmiennej

Zmienna (w Pythonie nazywana często po prostu nazwą), taka jak `a`, tworzona jest, kiedy kod pierwszy raz przypisuje jej wartość. Kolejne przypisania zmieniają wartość utworzonej wcześniej zmiennej. Z technicznego punktu widzenia wygląda to tak, że Python wykrywa niektóre nazwy zmiennych przed wykonaniem kodu, ale możemy to sobie wyobrazić tak, jakby początkowe przypisanie tworzyło zmienną.

Typ zmiennej

Zmienna nigdy nie ma dołączonych żadnych informacji o typie czy ograniczeniach z nim związanych. Pojęcie typu wiąże się z obiektami, a nie nazwami. Zmienne są z natury uniwersalne. Zawsze w danym momencie odnoszą się po prostu do określonego obiektu.

Użycie zmiennej

Kiedy zmienna pojawia się w wyrażeniu, jest natychmiast zastępowana przez obiekt, do którego się aktualnie odnosi, bez względu na to, czym by on nie był. Co więcej, wszystkie zmienne muszą mieć jawnie przypisane wartości, zanim będzie można ich użyć. Próba odwołania się do nieprzypisanej zmiennej kończy się błędem.

Podsumowując, zmienne tworzone są przy przypisaniu, mogą się odwoływać do dowolnego typu obiektu i muszą być przypisane przed wykonaniem referencji do nich. Oznacza to, że nigdy nie trzeba deklarować zmiennych wykorzystywanych w skrypcie, jednak przed ich uaktualnieniem niezbędna jest ich inicjalizacja. Przykładowo liczniki trzeba najpierw zainicjalizować wartością początkową (np. 0), aby można było je później inkrementować.

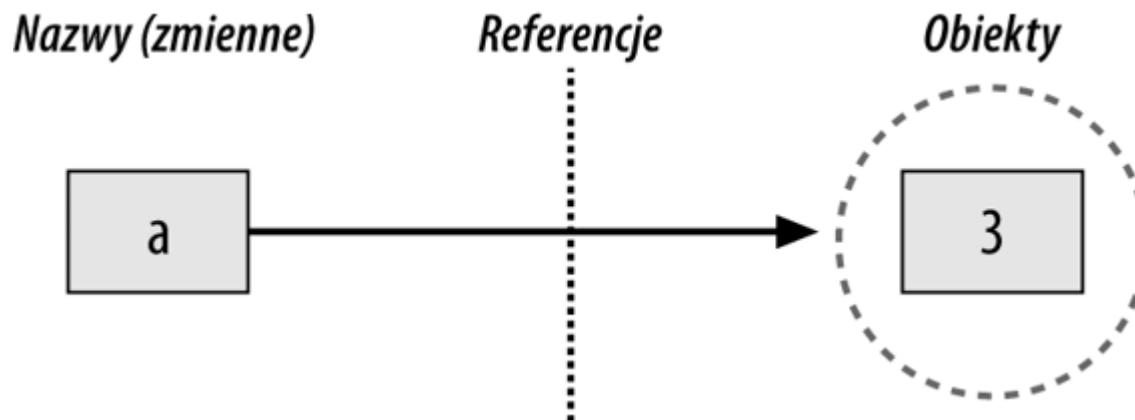
Model z typami dynamicznymi bardzo się różni od modelu typów w tradycyjnych językach programowania. Kiedy zaczynasz przygodę z Pythonem, najłatwiej jest zrozumieć typy dynamiczne, gdy jasno rozdziela się od siebie nazwy (zmienne) i obiekty. Na przykład, kiedy napiszemy taki kod:

```
>>> a = 3 #Przypisanie nazwy do obiektu
```

to przynajmniej z koncepcjonalnego punktu widzenia Python wykona trzy osobne kroki, by wykonać to żądanie. Kroki opisane poniżej odzwierciedlają działanie wszystkich przypisań w języku Python:

1. Utworzenie obiektu reprezentującego wartość 3.
2. Utworzenie zmiennej a, o ile jeszcze nie istnieje.
3. Połączenie zmiennej a z nowym obiektem 3.

Rezultatem będzie utworzenie „wewnętrz” Pythona struktury, która została schematycznie przedstawiona na rysunku 6.1. Jak łatwo zauważyć, zmienne i obiekty przechowywane są w różnych częściach pamięci i powiązane ze sobą połączeniami (takie połączenie widoczne jest na rysunku jako strzałka). Zmienne zawsze łączą się z obiektami i nigdy z innymi zmiennymi, jednak większe obiekty mogą łączyć się z innymi obiektami (jak na przykład obiekt listy łączący się z obiektami, które zawiera).



Rysunek 6.1. Zmienne i obiekty po wykonaniu przypisania `a = 3`. Zmienna `a` staje się referencją do obiektu 3. Wewnętrznie zmienna jest tak naprawdę wskaźnikiem do miejsca w pamięci zajmowanego przez obiekt i utworzonego przez wykonanie wyrażenia literatu 3

Te połączenia pomiędzy zmiennymi a obiektami nazywane są w Pythonie *referencjami*. Referencja to rodzaj powiązania zaimplementowanego jako wskaźnik w pamięci^[1]. Za każdym razem, gdy zmienne są używane (to znaczy korzysta się z referencji), Python automatycznie podąży połączeniem między zmienną a obiektem. Wszystko to jest o wiele prostsze, niż może wynikać z terminologii. A mówiąc konkretnie:

- *Zmienne* są wpisami w tabeli systemowej z miejscem na łącze do obiektów.
- *Obiekty* to fragmenty przydzielonej pamięci z ilością miejsca wystarczającą, by zmieścić wartości, które reprezentują.
- *Referencje* to wskaźniki między zmiennymi a obiektami, którymi automatycznie podąża Python.

Przynajmniej z koncepcyjnego punktu widzenia za każdym razem, gdy w skrypcie generowana jest nowa wartość poprzez wykonanie jakiegoś wyrażenia, Python tworzy *nowy obiekt* (czyli fragment pamięci) reprezentujący tę wartość. Wewnętrznie, ze względu na optymalizację, Python umieszcza pewne rodzaje niezmiennych obiektów w pamięci podręcznej i używa ich ponownie; jest tak na przykład w przypadku małych liczb całkowitych orazłańcuchów znaków (nie każde 0 jest tak naprawdę nowym fragmentem pamięci — więcej na ten temat później). Jednak z logicznego punktu widzenia działa to tak, jakby wartość wynikowa każdego wyrażenia była osobnym obiektem, a każdy obiekt był osobną częścią pamięci.

Z technicznego punktu widzenia obiekty mają nieco bardziej rozbudowaną strukturę niż tylko „tyle miejsca, by zmieścić swoją wartość”. Każdy obiekt ma również dwa standardowe pola nagłówków — *desygnator typu* wykorzystywany jest do oznaczenia typu obiektu, natomiast *licznik referencji* służy do ustalenia, kiedy obiekt można uwolnić. By zrozumieć, jakie znaczenie mają te dwa dodatkowe pola nagłówków, musimy przejść do kolejnego podrozdziału.

Typy powiązane są z obiektami, a nie ze zmiennymi

Aby zobaczyć, w którym miejscu pojawiają się typy, warto prześledzić, co się stanie, kiedy zmienną przypiszemy kilka razy.

```
>>> a = 3                                # Jest liczbą całkowitą
>>> a = 'mielonka'                         # Teraz jest łańcuchem znaków
>>> a = 1.23                               # A teraz liczbą
                                              zmiennoprzecinkową
```

Nie jest to kod typowy dla Pythona, ale działa — zmienna `a` na początku jest liczbą całkowitą, później staje się łańcuchem znaków, a na koniec staje się liczbą zmiennoprzecinkową. Ten przykład może wyglądać szczególnie dziwnie dla programistów języka C, ponieważ wygląda na to, jakby typ zmiennej `a` zmieniał się z liczby całkowitej na łańcuch znaków, kiedy wykona się instrukcję `a = 'mielonka'`.

Tak naprawdę dzieje się jednak coś innego. W Pythonie wszystko jest jeszcze prostsze. *Nazwy* (zmienne) nie mają typów; jak wspomniano wcześniej, typy powiązane są z obiektami, a nie z nazwami. W poprzednim listingu zmodyfikowaliśmy tylko zmienną `a`, tak by odnosiła się ona do innych obiektów. Ponieważ zmienne nie mają typu, nie zmodyfikowaliśmy tak naprawdę typu zmiennej `a` — zmienna ta odnosi się po prostu teraz do innego typu obiektu. Tak naprawdę jedynie, co można powiedzieć o zmiennych w Pythonie, to to, że odnoszą się do określonego obiektu w określonym momencie.

Obiekty wiedzą z kolei, jakiego są typu, ponieważ każdy obiekt zawiera pole nagłówka informujące o jego typie. Obiekt liczby całkowitej 3 zawiera na przykład wartość 3 i oprócz tego desygnator typu informujący Pythona, że obiekt ten jest liczbą całkowitą (a tak naprawdę wskaźnik do obiektu `int`, nazwy typu liczby całkowitej). Desygnator typu obiektu łańcucha znaków '`mielonka`' wskazuje z kolei na typ łańcucha znaków o nazwie `str`. Ponieważ obiekty znajdują swoje typy, zmienne nie muszą ich znać.

Powtarzając raz jeszcze: typy są w Pythonie powiązane z obiektami, a nie ze zmiennymi. W typowym kodzie dana zmienna będzie zazwyczaj zawierała referencję do jednego rodzaju obiektu. Ponieważ nie jest to jednak wymagane, wkrótce przekonasz się, że kod napisany w Pythonie zazwyczaj jest o wiele bardziej elastyczny od tego, do czego możesz być przyzwyczajony. Jeżeli będziemy dobrze wykorzystywać Pythona, kod przez nas tworzony może automatycznie działać na wielu typach obiektów.

Wspomniałem wyżej, że obiekty zawierają dwa pola nagłówków — desygnator typu i licznik referencji. Aby zrozumieć znaczenie tego drugiego pola, musimy przyjrzeć się temu, co dzieje się na końcu życia obiektu.

Obiekty są uwalniane

W kodzie wyżej przypisaliśmy zmienną `a` do trzech różnych typów obiektów. Co się jednak po kolejnym przypisaniu dzieje z wartością, do której odnosił się obiekt? Co na przykład stanie się z obiektem 3 po wykonaniu poniższej instrukcji?

```
>>> a = 3  
>>> a = 'mielonka'
```

Odpowiedź jest następująca: w Pythonie za każdym razem, gdy zmienna przypisywana jest do nowego obiektu, miejsce zajmowane przez poprzedni obiekt jest uwalniane (o ile nie odwołuje się do niego żadna inna nazwa lub obiekt). Takie automatyczne zwalnianie przestrzeni obiektu nazywane jest *czyszczeniem pamięci* (ang. *garbage collection*).

Aby zilustrować ten mechanizm, rozważmy poniższy przykład, który przyporządkowuje zmiennej `x` inny obiekt przy każdej instrukcji przypisania.

```
>>> x = 42  
>>> x = ' żywopłot' # Uwolnienie liczby 42 (o ile  
nie ma innych referencji)  
>>> x = 3.1415 # Uwolnienie łańcucha znaków  
' żywopłot'  
>>> x = [1,2,3] # Uwolnienie liczby 3.1415
```

Po pierwsze zauważ, że zmienna `x` za każdym razem odnosi się do innego typu obiektu. I znowu, choć tak naprawdę wcale tak nie jest, możesz odnieść wrażenie, jakby typ zmiennej `x` z czasem się zmieniał. Pamiętaj, że w Pythonie typy powiązane są jednak z obiektami, a nie z nazwami. Ponieważ zmienne są uniwersalnymi referencjami do obiektów, taki kod działa w naturalny sposób.

Po drugie, warto zwrócić uwagę na to, że referencje do obiektów są po drodze porzucane. Za każdym razem, gdy do zmiennej `x` przypisujemy nowy obiekt, Python zwalnia miejsce zajmowane przez poprzedni. Kiedy na przykład do zmiennej `x` przypisujemy łańcuchów znaków ' `żywopłot`', obiekt 42 jest natychmiast zwalniany (zakładając, że nie istnieje żadna inna referencja do niego). Tym samym miejsce zajmowane przez ten obiekt jest automatycznie wrzucane z powrotem do puli wolnego miejsca, które może być użyte przez kolejny obiekt.

Wewnętrznie Python wykonuje to wszystko, przechowując w każdym obiekcie licznik, który śledzi liczbę referencji wskazujących na ten obiekt. W momencie gdy licznik spadnie do zera, miejsce zajmowane przez obiekt w pamięci jest automatycznie zwalniane. W poprzednim listingu zakładamy, że za każdym razem, gdy do zmiennej `x` przypisujemy nowy obiekt, licznik referencji poprzedniego obiektu spada do zera, co sprawia, że jest on zwalniany.

Kilka słów o czyszczeniu pamięci w języku Python

Z technicznego punktu widzenia mechanizm czyszczenia pamięci w Pythonie oparty jest przede wszystkim na *licznikach referencji*, tak jak o tym pisaliśmy powyżej, ale zawiera również komponent wykrywający i zwalniający obiekty z *referencjami cyklicznymi*.

Komponent ten można wyłączyć, jeżeli jesteś pewny, że Twój kod nie tworzy cykli, jednak domyślnie jest on włączony.

Referencje (odwołania) cykliczne to klasyczny problem, z jakim borykają się mechanizmy czyszczenia pamięci. Ponieważ referencje zaimplementowane są w postaci wskaźników, obiekt może odwoływać się do samego siebie bądź też do obiektu, który zawiera do niego referencję. Ćwiczenie 6. znajdujące się na końcu pierwszej części książki oraz jego rozwiązanie z dodatku D pokazują na przykład, jak można utworzyć odwołanie cykliczne, zagnieżdżając referencję do listy wewnętrznej niej samej (np. `L.append(L)`). To samo zjawisko może wystąpić w przypadku przypisań do atrybutów obiektów tworzonych z klas definiowanych przez użytkownika. Choć jest to niezwykle rzadkie, sytuacje takie muszą być traktowane w specjalny sposób, ponieważ liczniki referencji dla takich obiektów nigdy nie spadną do zera.

Szczegółowe informacje na temat wykrywacza odwołań cyklicznych w Pythonie można znaleźć w dokumentacji modułu `gc` dostępnej w podręczniku biblioteki standardowej Pythona. Wystarczy jednak powiedzieć, że mechanizm zarządzania pamięcią został zaimplementowany w Pythonie przez programistów mających ogromne doświadczenie w tworzeniu takich rozwiązań.

Warto także zauważyć, że opis mechanizmu czyszczenia pamięci, zamieszczony w tym rozdziale, odnosi się jedynie do standardowej implementacji CPython. Omawiane w rozdziale 2. dystrybucje Jython, IronPython czy PyPy mogą korzystać z innych mechanizmów, choć rezultat ich działania będzie podobny — niewykorzystywane miejsce jest automatycznie zwalniane.

Najbardziej zauważalną zaletą czyszczenia pamięci jest to, że można swobodnie korzystać z obiektów bez konieczności troszczenia się w skrypcie o zwalnianie miejsca w pamięci. Python automatycznie čzyści za nas nieużywaną przestrzeń w miarę wykonywania programu. W praktyce eliminuje to dużą ilość kodu zarządzającego alokacją pamięci, zwłaszcza w porównaniu z językami niższego poziomu, takimi jak C czy C++.

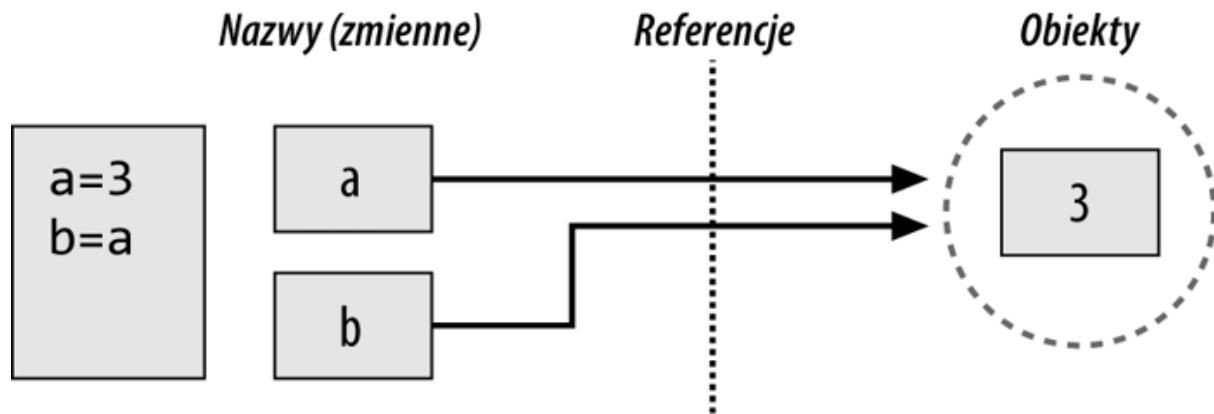
Referencje współdzielone

Dotychczas widzieliśmy, co dzieje się, kiedy do jednej zmiennej przypisujemy referencje do obiektów. Teraz wprowadzimy do naszego kodu kolejną zmienną i sprawdzimy, co stanie się z nazwami i obiektami.

```
>>> a = 3  
>>> b = a
```

Wpisanie tych dwóch instrukcji generuje sytuację ujętą na rysunku 6.2. Drugi wiersz sprawia, że Python tworzy zmienną `b`. Zmienna `a` jest tutaj użyta, ale nie przypisana, dlatego zostaje zastąpiona obiektem, do którego się odnosi (3), natomiast `b` zawiera odniesienie do tego obiektu. Rezultat jest taki, że zmienne `a` i `b` odnoszą się do tego samego obiektu (czyli wskazują na ten sam fragment pamięci). Jest to w Pythonie znane pod nazwą *referencji współdzielonych*.

(ang. *shared references*) i oznacza, że w praktyce wiele zmiennych może się odnosić do tego samego obiektu (czyli inaczej mówiąc, będzie wskazywać na ten sam obszar pamięci, zajmowany przez dany obiekt).



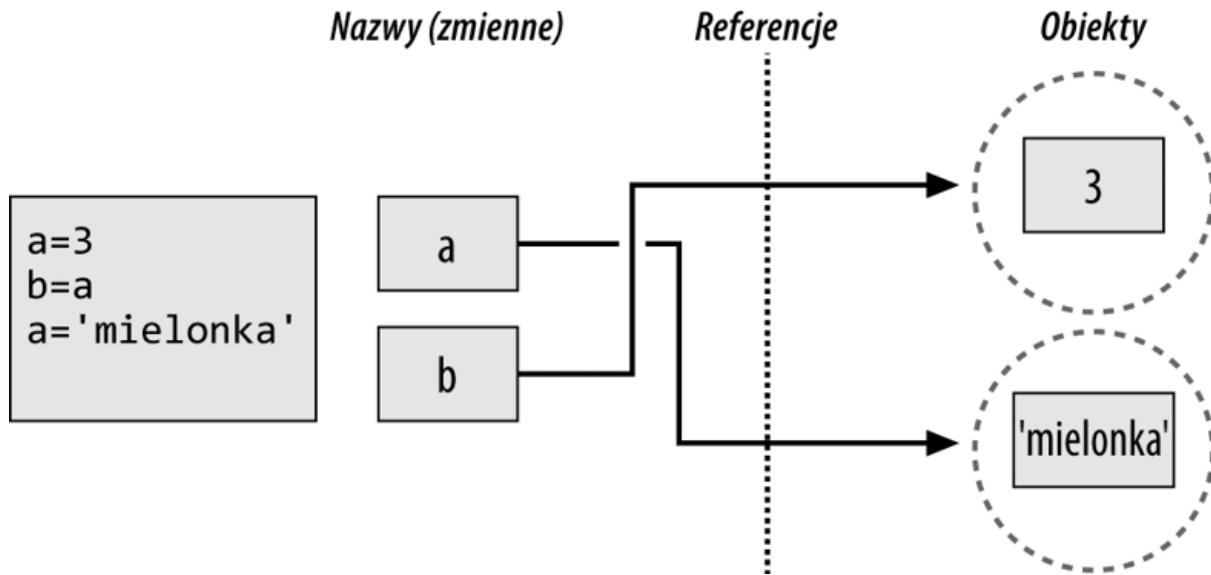
Rysunek 6.2. Nazwy (zmienne) i obiekty po przypisaniu $b = a$. Zmienna b staje się referencją do obiektu 3. Wewnętrznie zmieniona jest tak naprawdę wskaźnikem do miejsca w pamięci zajmowanego przez obiekt i utworzonego przez wykonanie wyrażenia literatu 3

Taki scenariusz z wieloma nazwami (zmiennymi) odwołującymi się do tego samego obiektu jest w Pythonie zwykle nazywany *współdzieloną referencją* (a czasami po prostu *współdzielonym obiektem*). Zauważ, że nazwy a i b nie są ze sobą bezpośrednio powiązane, gdy tak się dzieje; w rzeczywistości w Pythonie nie ma możliwości połączenia zmiennej z inną zmienną; zamiast tego obie zmienne wskazują raczej na ten sam obiekt za pomocą swoich referencji.

Teraz założymy, że rozszerzamy ten interaktywny kod o kolejną instrukcję:

```
>>> a = 3
>>> b = a
>>> a = 'mielonka'
```

Tak jak w przypadku wszystkich przypisań w Pythonie, instrukcja ta tworzy nowy obiekt reprezentujący wartość 'mielonka' i ustawia zmienną a w taki sposób, by odnosiła się do tego nowego obiektu. Nie zmienia jednak wartości zmiennej b — nadal odnosi się ona do oryginalnego obiektu, czyli liczby całkowitej 3. Struktura referencji będzie teraz zatem taka, jak na rysunku 6.3.



Rysunek 6.3. Nazwy (zmienne) i obiekty po przypisaniu `a = 'mielonka'`. Po wykonaniu wyrażenia z literałem 'mielonka' zmienna `a` staje się referencją do nowego obiektu (miejsc w pamięci), natomiast zmienna `b` nadal odwołuje się do oryginalnego obiektu — liczby 3. Ponieważ to przypisanie nie jest modyfikacją obiektu 3 w miejscu, modyfikuje ono jedynie zmienną `a`, a nie `b`

To samo stałoby się, gdybyśmy zamiast tego zmienili `b` na 'mielonka' — przypisanie zmieniłoby jedynie zmienną `b`, a nie `a`. Takie zachowanie pojawia się również wtedy, gdy nie ma żadnej różnicy w zakresie typów. Rozważmy na przykład trzy poniższe instrukcje.

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

W tej sekwencji przypisań dzieje się to samo. Python przypisuje do zmiennej `a` obiekt 3, następnie `b` staje się referencją do tego samego obiektu co `a` (jak na rysunku 6.2). Jak wcześniej, ostatnie przypisanie zmienia `a` na zupełnie inny obiekt (w tym przypadku liczbę całkowitą 5, będącą wynikiem wyrażenia z operatorem `+`). Nie modyfikuje to jednak zmiennej `b`. Tak naprawdę nie da się nadpisać wartości obiektu 3 — tak jak pisaliśmy w rozdziale 4., liczby całkowite są niezmienne i tym samym nie można ich modyfikować w miejscu.

Można to sobie wyobrazić w taki sposób, że w przeciwieństwie do niektórych języków programowania w Pythonie zmienne zawsze są wskaźnikami do obiektów, a nie podpisami zmieniających się miejsc w pamięci. Nadanie zmiennej nowej wartości nie modyfikuje oryginalnych obiektów, ale raczej sprawia, że zmienna odnosi się do całkowicie innego obiektu. Rezultat jest taki, że przypisanie do zmiennej może wpływać jedynie na tę jedną zmienną, do której coś się przypisuje. Kiedy do równania dodamy również obiekty zmienne i zmiany w miejscu, obraz ten nieco się zmieni. Aby się o tym przekonać, przejdźmy dalej.

Referencje współdzielone a modyfikacje w miejscu

Jak zobaczymy w kolejnych rozdziałach tej części książki, w Pythonie istnieją obiekty i operacje modyfikujące obiekt *w miejscu* — *mutowalne* typy Pythona, w tym listy, słowniki i zbiory. Na przykład przypisanie do pozycji przesunięcia w liście zmienia listę w miejscu, nie generuje nowego obiektu listy. W przypadku obiektów obsługujących zmiany tego rodzaju należy bardziej

uważać na referencje współdzielone, ponieważ zmiana dokonana w jednym miejscu może wpływać na inne obiekty.

Choć w tym miejscu książki musisz mi jeszcze trochę uwierzyć na słowo, pamiętaj, że opisane wyżej rozróżnienie może mieć duże znaczenie dla Twoich programów. W przypadku obiektów, które obsługują modyfikacje w miejscu, musisz być bardziej świadomy wspólnych odwołań, ponieważ zmiana jednej nazwy może mieć wpływ na inne. W przeciwnym razie możesz odnieść wrażenie, że Twoje obiekty zmieniają się bez jakiegoś wyraźnego powodu. Biorąc pod uwagę, że wszystkie przypisania są oparte na referencjach (w tym przekazywanie argumentów funkcji), takie sytuacje mogą się zdarzać dosyć często.

Aby zilustrować tę kwestię, przyjrzyjmy się raz jeszcze obiektom listy wprowadzonym w rozdziale 4. Warto przypomnieć, że listy obsługujące przypisanie w miejscu do określonych pozycji są po prostu kolekcjami innych obiektów, zakodowanymi w nawiasach kwadratowych.

```
>>> L1 = [2, 3, 4]  
>>> L2 = L1
```

L1 to lista zawierająca obiekty 2, 3 i 4. Dostęp do obiektów znajdujących się wewnątrz tej listy można uzyskać za pomocą ich pozycji, dzięki czemu L1[0] odnosi się do obiektu 2, pierwszego elementu listy L1. Listy są oczywiście obiektem same w sobie, podobnie jak liczby całkowite i łańcuchy znaków. Po wykonaniu dwóch powyższych instrukcji przypisania L1 i L2 odnoszą się do tego samego, współdzielonego obiektu, podobnie jak a i b z poprzedniego przykładu (zobacz rysunek 6.2). Powiedzmy, że teraz znów, tak jak poprzednio, rozszerzymy ten kod o kolejne przypisanie:

```
>>> L1 = 24
```

Przypisanie to sprawia, że L1 będzie teraz referencją do innego obiektu, natomiast L2 nadal jest referencją do oryginalnej listy. Gdybyśmy nieco zmienili składnię tej instrukcji, miałaby ona zupełnie inny efekt.

```
>>> L1 = [2, 3, 4]                      # Zmienny obiekt  
>>> L2 = L1                            # Referencja do tego samego  
obiektu  
>>> L1[0] = 24                          # Zmiana w miejscu  
>>> L1                                  # Lista L1 zmienia się  
[24, 3, 4]  
>>> L2                                  # Tak samo lista L2!  
[24, 3, 4]
```

Tym razem nie zmodyfikowaliśmy tak naprawdę samej listy L1, a jedynie komponent *obiektu*, do którego odnosi się L1. Ten rodzaj zmiany powoduje nadpisanie części obiektu listy w miejscu. Ponieważ obiekt listy jest współdzielony przez różne zmienne (które się do niego odnoszą), modyfikacja tego typu nie ma zastosowania jedynie do L1. Trzeba pamiętać, że zmiany tego typu mogą mieć wpływ na inne części naszego programu. W powyższym przykładzie widać to w L2, gdyż zmienna ta odnosi się do tego samego obiektu co L1. I choć nie modyfikowaliśmy także L2, wartość tej zmiennej jest już inna, ponieważ odnosi się ona do obiektu, który został zmodyfikowany w miejscu.

Takie zachowanie występuje wyłącznie w przypadku obiektów mutowalnych, które mogą być modyfikowane w miejscu, i jest zazwyczaj czymś pożądanym, choć należy być go świadomym, tak by zmiany można było przewidzieć. Jest to zachowanie domyślne. Jeżeli jednak nie jest przez nas pożądane, można zażądać, by Python *kopiował* obiekty, zamiast tworzyć do nich referencje. Istnieje kilka sposobów skopowania listy, w tym wykorzystanie wbudowanej funkcji

`list`, a także modułu `copy` z biblioteki standardowej. Chyba najczęściej stosowaną techniką jest sporządzenie wycinka z całej listy (więcej informacji na temat wycinków znajduje się w rozdziałach 4. oraz 7.).

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]
# Utworzenie kopii L1 (lub
list(L1), copy.copy(L1) itp.)
>>> L1[0] = 24
>>> L1
[24, 3, 4]
>>> L2
# L2 się nie zmienia
[2, 3, 4]
```

W powyższym kodzie zmiany wprowadzone do `L1` nie są przenoszone do `L2`, ponieważ zmenna `L2` odnosi się do kopii obiektu `L1`, a nie do oryginału. Oznacza to, że obie zmienne odnoszą się do dwóch różnych fragmentów pamięci.

Warto podkreślić, że technika z wycinkami nie będzie działała na innych mutowalnych typach podstawowych, czyli słownikach oraz zbiorach, ponieważ nie są one sekwencjami. W celu skopiowania słownika lub zbioru powinieneś użyć wywołania ich metody `X.copy()`; warto zauważać, że począwszy od wersji 3.3 Pythona, listy również posiadają taką metodę. W module `copy` z biblioteki standardowej dostępna jest funkcja pozwalająca na tworzenie kopii obiektów dowolnego typu, a także funkcja kopiąca zagnieżdżoną strukturę obiektów (na przykład słownik z zagnieżdżonymi listami).

```
import copy
X = copy.copy(Y)
# Wykonanie "pływkiej" kopii
dowolnego obiektu Y
X = copy.deepcopy(Y)
# Wykonanie "głębokiej" kopii
dowolnego obiektu Y
# – skopiowanie wszystkich
elementów zagnieżdżonych
```

Listy i słowniki, a także koncepcję współdzielonych referencji oraz kopii omówimy bardziej szczegółowo w rozdziałach 8. i 9. Na razie powinieneś zapamiętać, że obiekty mutowalne (czyli takie, które można modyfikować w miejscu) są zawsze podatne na tego rodzaju efekty. W Pythonie dotyczy to list, słowników, zbiorów i pewnych obiektów definiowanych za pomocą instrukcji `class`. Jeżeli nie jest to pożądane zachowanie, zawsze możesz po prostu utworzyć kopię takiego obiektu.

Referencje współdzielone a równość

Aby wszystko było jasne, warto wspomnieć, że czyszczenie pamięci opisane wcześniej może dla niektórych typów obiektów być bardziej konceptualne niż dosłowne. Rozważmy poniższe instrukcje:

```
>>> x = 42
>>> x = ' żywopłot'
# Uwolnić teraz obiekt 42?
```

Ponieważ Python umieszcza małe liczby całkowite oraz łańcuchy znaków w pamięci podręcznej, aby użyć ich ponownie, obiekt 42 prawdopodobnie nie zostanie faktycznie uwolniony. Zamiast tego pozostałe raczej w tabeli systemowej, tak by można go było wykorzystać następnym

razem, gdy w kodzie użyjemy liczby 42. Większość typów obiektów jest jednak uwalniana od razu po tym, jak znika ostatnia referencja do nich. W przypadku tych, które nie są uwalniane, mechanizm umieszczania w pamięci podręcznej nie ma znaczenia dla naszego kodu.

Przykładowo ze względu na model referencji Pythona istnieją dwa różne sposoby sprawdzania równości w programach napisanych w tym języku. Aby to zademonstrować, utworzymy współdzieloną referencję.

```
>>> L = [1, 2, 3]
>>> M = L                                # M i L odnoszą się do jednego
obiektu
>>> L == M                                # Ta sama wartość
True
>>> L is M                                # Ten sam obiekt
True
```

Pierwsza z technik zaprezentowanych powyżej, operator `==`, sprawdza, czy dwa obiekty, do których odnoszą się zmienne, mają tę samą wartość. Metoda ta jest prawie zawsze wykorzystywana w sprawdzaniu równości w Pythonie. Druga metoda, operator `is`, sprawdza zamiast tego *identyczność* obiektów — zwraca `True` tylko wtedy, gdy obie zmienne odnoszą się do dokładnie tego samego obiektu, dlatego jest o wiele mocniejszą formą sprawdzania równości i w praktyce rzadko jest stosowana w programach.

Tak naprawdę operator `is` sprawdza po prostu wskaźniki implementujące referencje, dzięki czemu jest w stanie w razie potrzeby wykrywać w kodzie współdzielone referencje. Zwraca `False`, jeżeli zmienne odnoszą się do odpowiadających sobie, ale jednak różnych obiektów, tak jak wtedy, gdy wykonamy dla nich różne wyrażenia z literałami.

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]                          # M i L odnoszą się do różnych
obiektów
>>> L == M                                # Te same wartości
True
>>> L is M                                # Różne obiekty
False
```

Sprawdźmy teraz, co stanie się, gdy te same operacje wykonamy na małych liczbach.

```
>>> X = 42
>>> Y = 42                                # Powinny być dwoma różnymi
obiektami
>>> X == Y                                 # A jednak to jeden obiekt
True
>>> X is Y
(pamięć podręczna działa)!
```

True

W powyższym kodzie `X` i `Y` powinny mieć tę samą wartość (operator `==`), jednak nie powinny być tym samym obiektem (operator `is`), ponieważ wykonaliśmy dwa osobne wyrażenia z literałami

(42). Ponieważ jednak małe liczby i łańcuchy znaków umieszczane są w pamięci podręcznej i używane ponownie, okazuje się, że obie zmienne odnoszą się do tego samego obiektu.

Gdybyśmy naprawdę chcieli przyjrzeć się temu, jak to działa, zawsze możemy zapytać Pythona, ile jest referencji do określonego obiektu. Funkcja `getrefcount` z modułu standardowego `sys` zwraca licznik referencji obiektu. Na przykład, kiedy w środowisku IDLE zapytałem o liczbę referencji do obiektu liczby całkowitej 1, Python zwrócił wynik 647 (większość referencji pochodzi z kodu systemowego środowiska IDLE, a nie mojego własnego, zatem sam Python również musi korzystać z wielu odwołań do tej liczby).

```
>>> import sys  
>>> sys.getrefcount(1) # 647 wskaźników do tego miejsca  
w pamięci  
647
```

Umieszczanie obiektów w pamięci podręcznej i ponowne ich użycie nie ma znaczenia dla naszego kodu (o ile oczywiście nie sprawdzamy równości za pomocą `is`). Ponieważ niemutowalnych liczb i łańcuchów znaków nie można modyfikować w miejscu, nie ma znaczenia, ile jest referencji do tego samego obiektu — każde odwołanie będzie zawsze wskazywało tę samą, niezmodyfikowaną wartość. Zachowanie to odzwierciedla jednak jeden z wielu sposobów optymalizacji modelu Pythona pod kątem szybkości działania.

Typy dynamiczne są wszędzie

Oczywiście aby używać Pythona, nie trzeba tak naprawdę rysować diagramów z nazwami zmiennych, obiektami i strzałkami. Na początek jednak prześledzenie struktury referencji pomaga często zrozumieć pozornie „niezrozumiałe” zachowania programu. Jeżeli na przykład obiekt mutowalny zostanie zmodyfikowany w czasie przekazywania w programie, jest duża szansa, że trafiliśmy na zagadnienia omówione w niniejszym rozdziale.

Co więcej, nawet jeżeli typy dynamiczne wydają się w tej chwili nieco abstrakcyjne, z pewnością wkrótce uda Ci się przyswoić tę koncepcję. Ponieważ w Pythonie *wszystko* zdaje się działać przez przypisanie i referencje, podstawowe zrozumienie tego modelu przydaje się w wielu różnych kontekstach. Jak się niebawem przekonasz, tak samo działa to w instrukcjach przypisania, argumentach funkcji, zmiennych pętli `for`, importowaniu modułów, atrybutach klas i innych komponentach. Dobra wiadomość jest taka, że w Pythonie istnieje tylko *jeden* model przypisania. Kiedy zrozumiesz typy dynamiczne, z pewnością zauważysz, że w całym języku działają one w ten sam sposób.

Z praktycznego punktu widzenia typy dynamiczne oznaczają mniej kodu, który musimy napisać. Co jednakowo ważne, typy dynamiczne są również podstawą *polimorfizmu* Pythona, czyli koncepcji przedstawionej w rozdziale 4., do której powrócimy w dalszej części książki. Ponieważ w Pythonie typy nie są ograniczane, jego kod jest zarówno zwięzły, jak i bardzo elastyczny. Jak zobaczymy niebawem, kiedy się je dobrze wykorzystuje, typy dynamiczne i polimorfizm tworzą kod, który wraz z rozwojem systemu automatycznie adaptuje się do nowych wymagań.

„Słabe” referencje

Od czasu do czasu w świecie Pythona możesz się spotkać z określeniem „słabe odnośniki” bądź „słabe referencje”. Jest to dosyć zaawansowane narzędzie, ale jest mocno związane z modelem referencyjnym, który tutaj badaliśmy, i podobnie jak operator `is`, nie może być bez niego zrozumiane.

Krótko mówiąc, słabe odwołanie, zaimplementowane w module standardowej biblioteki `weakref`, jest odwołaniem do obiektu, które samo w sobie nie zapobiega jego usunięciu

przez mechanizm czyszczenia pamięci (ang. *garbage collection*). Jeżeli ostatnie pozostające odniesienia do obiektu są słabymi odniesieniami, obiekt jest czyszczony, a słabe odniesienia do niego są automatycznie usuwane (lub w inny sposób powiadamiane).

Może to być przydatne na przykład w opartych na słownikach buforach dużych obiektów; w przeciwnym razie sama referencja w pamięci podręcznej utrzymywałaby obiekt w pamięci na czas nieokreślony. Nie zmienia to jednak w niczym faktu, że jest to po prostu specjalnego przeznaczenia rozszerzenie modelu referencyjnego. Więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji bibliotek Pythona.

Podsumowanie rozdziału

W niniejszym rozdziale szerzej przyjrzaliśmy się modelowi typów dynamicznych Pythona, czyli temu, w jaki sposób Python automatycznie śledzi typy obiektów, zamiast wymagać od nas deklarowania ich w skryptach. Przy okazji dowiedziałeś się, jak zmienne i obiekty w Pythonie są powiązane referencjami. Omówiliśmy również kwestię czyszczenia pamięci, pokazaliśmy, jak współdzielone referencje do obiektów mogą wpływać na wiele zmiennych, a także jaki wpływ mają na zagadnienie równości w Pythonie.

Ponieważ w Pythonie używany jest tylko jeden model przypisania i ponieważ przypisania pojawiają się w tym języku niemal wszędzie, zrozumienie tego modelu przed przejściem do dalszego omawiania języka jest bardzo istotne. Quiz znajdujący się na końcu rozdziału pomoże usystematyzować Ci niektóre z przedstawionych tutaj koncepcji. Potem, w następnym rozdziale, powrócimy do omawiania kolejnych typów obiektów — tym razem będą to łańcuchy znaków.

Sprawdź swoją wiedzę — quiz

1. Rozważmy trzy poniższe instrukcje. Czy zmieniają one wartość zmiennej A?

```
A = "mielonka"  
B = A  
B = "żywopłot"
```

2. Rozważmy trzy poniższe instrukcje. Czy zmieniają one wartość zmiennej A?

```
A = ["mielonka"]  
B = A  
B[0] = "żywopłot"
```

3. Czy tym razem wartość A się zmieni?

```
A = ["mielonka"]  
B = A[:]  
B[0] = "żywopłot"
```

Sprawdź swoją wiedzę — odpowiedzi

1. Nie. A nadal ma wartość "mielonka". Kiedy B przypiszemy do łańcucha znaków " żywopłot ", zmienna B jest modyfikowana tak, aby wskazywała teraz na ten nowy łańcuch znaków. A i B początkowo współdzielą (to znaczy odnoszą się, wskazują) ten sam łańcuch znaków " mielonka ", jednak obie zmienne nigdy nie są ze sobą w żaden sposób połączone. Ustawienie B na inny obiekt nie ma zatem wpływu na A. Tak samo byłoby, gdyby ostatnią instrukcją było `B = B + ' żywopłot '` — konkatenacja utworzyłaby z wyniku nowy obiekt, który zostałby następnie przypisany jedynie do B. Nigdy nie możemy nadpisać łańcucha znaków (albo liczby czy krotki) w miejscu, ponieważ ciągi znaków są niemutowalne.
2. Tak. A ma teraz wartość [" żywopłot "]. Z technicznego punktu widzenia nie zmieniliśmy ani A, ani B. Zamiast tego zmodyfikowaliśmy część obiektu, do którego odnoszą się obie zmienne, nadpisując ten obiekt w miejscu poprzez zmienną B. Ponieważ A odnosi się do tego samego obiektu co B, zmiana ta odzwierciedlana jest również w A.
3. Nie. A nadal ma wartość [" mielonka "]. Przypisanie w miejscu za pomocą B nie ma tym razem żadnego wpływu na A, ponieważ wyrażenie z wycinkiem sporządziło kopię obiektu listy przed przypisaniem go do B. Po drugiej instrukcji przypisania istnieją dwa różne obiekty listy o tej samej wartości (w Pythonie mówimy, że są one sobie równe `(==)`, ale nie są tożsame `(is)`). Trzecia instrukcja zmienia wartość obiektu listy, na który wskazuje zmienna B, jednak już nie tego, na który wskazuje A.

[1] Osoby znające język C mogą skojarzyć referencje w Pythonie ze wskaźnikami (adresami w pamięci) języka C. Tak naprawdę referencje zaimplementowane są jako wskaźniki i często pełnią te same role, w szczególności w przypadku obiektów, które mogą być modyfikowane w miejscu (więcej na ten temat później). Ponieważ jednak referencje są zawsze automatycznie usuwane po użyciu, nie można z nimi samymi zrobić nic przydatnego — jest to cecha, która eliminuje dużą kategorię błędów występujących w języku C. Referencje w Pythonie można sobie wyobrazić jako wskaźniki typu „void *” z języka C, które przy użyciu są automatycznie śledzone.

Rozdział 7. Łańcuchy znaków

Do tej pory zdążyliśmy omówić zagadnienia związane z typami liczbowymi i modelem typowania dynamicznego przyjętym w Pythonie. Kolejnym podstawowym typem, jakim zajmiemy się przy okazji omawiania wbudowanych typów obiektów, jest *łańcuch znaków* (ang. *string*) — uporządkowana kolekcja znaków wykorzystywana do przechowywania i reprezentowania informacji tekstowych oraz opartych na tekstowym zapisie ciągów bajtowych. Z łańcuchami znaków spotkaliśmy się już krótko w rozdziale 4. Teraz omówimy je bardziej szczegółowo, uzupełniając pewne informacje, które wcześniej pominęliśmy.

Co znajdziesz w tym rozdziale

Zanim zaczniemy, chciałbym również wyjaśnić, czym się tutaj nie będziemy zajmować. Rozdział 4. zawiera krótkie omówienie ciągów znaków i plików *Unicode*, które pozwalają Ci na radzenie sobie z tekstami innymi niż ASCII. Unicode jest kluczowym narzędziem dla niektórych programistów, szczególnie tych, którzy realizują zadania związane z domeną internetową — tekst w formacie Unicode można spotkać na przykład na stronach internetowych, w treści i nagłówkach wiadomości e-mail, transferach FTP, graficznych interfejsach użytkownika, plikach HTML, XML czy JSON i wielu innych narzędziach i usługach sieciowych.

Nie zmienia to jednak w niczym faktu, że Unicode może być trudnym tematem dla początkujących programistów dopiero rozpoczynających swoją działalność, a wielu (a może nawet większość) programistów używających Pythona pracuje w stanie błogiej niewiedzy bądź bardzo słabej znajomości zagadnień związanych z Unicode. Z tego względu w naszej książce przenieśliśmy zdecydowaną większość takich tematów do rozdziału 37. (część VII „Zagadnienia zaawansowane”, a w tym rozdziale skoncentrujemy się głównie na podstawowych zagadnieniach związanych z łańcuchami znaków.

Oznacza to, że ten rozdział opowiada tylko część historii związanych z ciągami znaków w Pythonie — część, której używa większość skryptów i którą powinna znać większość programistów. Zajmiemy się tutaj podstawowym typem łańcuchów znaków, które obsługują tekst ASCII i działają tak samo niezależnie od używanej wersji Pythona. Pomimo tego celowo ograniczonego zakresu wszystko, czego się tutaj nauczysz, będzie miało bezpośrednie zastosowanie również do przetwarzania tekstu Unicode, ponieważ w Pythonie 3.x typ `str` obsługuje także Unicode, a osobny typ `unicode` działa niemal identycznie jak `str` w wersji 2.x.

Unicode — krótka historia

Dla użytkowników, którym zależy na korzystaniu z Unicode, chciałbym również przedstawić krótkie podsumowanie jego wpływu i zostawić kilka wskazówek do dalszych, samodzielnych badań. Z formalnego punktu widzenia ASCII jest prostą formą tekstu Unicode, ale jednocześnie tylko jednym z wielu możliwych schematów kodowania i alfabetów. Teksty ze źródeł nieangielszczyznych mogą używać bardzo różnych znaków i mogą być kodowane na bardzo różne sposoby przy zapisywaniu w plikach.

Jak mogłeś się przekonać w rozdziale 4., Python rozwiązuje ten problem, rozróżniając pomiędzy danymi tekstowymi i binarnymi, z odrębnymi typami obiektów łańcuchowych i interfejsami

plików dla każdych z nich. Sposób obsługi różni się w zależności od wersji języka Python:

- W *Pythonie 3.x* występują trzy typy łańcuchów znaków: `str` używany jest dla tekstu Unicode (w tym ASCII), `bytes` jest używany dla danych binarnych (w tym tekstu zakodowanego), a `bytearray` jest mutowalną odmianą typu `bytes`. Pliki używane są w dwóch trybach: *tekstowym*, który reprezentuje zawartość jako ciągi znaków typu `str` i implementuje kodowanie Unicode, oraz *binarnym*, który zajmuje się surowymi ciągami bajtów i nie dokonuje translacji danych.
- W *Pythonie 2.x* ciągi typu `unicode` reprezentują tekst Unicode, ciągi znaków typu `str` obsługują zarówno tekst 8-bitowy, jak i dane binarne, a typ `bytearray` jest dostępny w wersji 2.6 i nowszych jako backport z wersji 3.x. Normalna zawartość plików to po prostu ciągi bajtów reprezentowane jako łańcuchy `str`, ale moduł `codecs` pozwala na otwieranie plików tekstowych Unicode, obsługuje różne formaty kodowania i prezentuje zawartość plików jako obiekty `unicode`.

Kiedy będziesz musiał użyć Unicode, przekonasz się, że mimo różnic w wersjach rozszerzenie jest stosunkowo niewielkie — gdy tekst zostanie już załadowany do pamięci, traktowany jest w języku Python jako zwykły łańcuch znaków, obsługujący wszystkie podstawowe elementy, które będziemy omawiać w tym rozdziale. W rzeczywistości podstawowa różnica między zwykłym tekstem a Unicode często sprowadza się tylko do wprowadzenia dodatkowego etapu kodowania (bądź dekodowania), niezbędnego do odpowiednio zapisywania i odczytywania takich danych do i z plików — poza tym w dużej mierze jest to po prostu zwykłe przetwarzanie ciągów znaków.

I znów, ponieważ większość programistów nie musi się na zapas przejmować zagadnieniami Unicode, większość szczegółów przeniosłem do rozdziału 37. Kiedy będziesz już gotowy, aby dowiedzieć się czegoś więcej o zaawansowanych koncepcjach ciągów, powinieneś zapoznać się podstawami omówionymi w tym rozdziale, a następnie zatrzymać się zarazem do krótkiego streszczenia w rozdziale 4., jak i pełnego omówienia Unicode i ciągów bajtowych w rozdziale 37.

W tym rozdziale skupimy się na podstawowym typie łańcuchów znaków i powiązanych z nim operacjach. Jak się przekonasz, techniki, które tu omówimy, odnoszą się również bezpośrednio do bardziej zaawansowanych typów ciągów znaków używanych w Pythonie.

Łańcuchy znaków — podstawy

Z funkcjonalnego punktu widzenia łańcuchy znaków można wykorzystać do reprezentowania wszystkiego, co można zakodować w postaci tekstowej lub bajtowej. W dziedzinie tekstów obejmuje to między innymi symbole i słowa (na przykład Twoje imię), pliki tekstowe załadowane do pamięci, adresy internetowe czy kod źródłowy programów napisanych w Pythonie. Ciągów znaków można również używać do przechowywania surowych danych bajtowych używanych na przykład w plikach multimedialnych i transferach sieciowych, a także zakodowanych i zdekodowanych tekstów w formacie Unicode innym niż ASCII, używanych w programach przeznaczonych do użytku w różnych krajach.

Łańcuchy znaków występują również w innych językach programowania. W Pythonie pełnią one tę samą rolę co tablice znaków z języków takich, jak C, jednak w pewnym sensie są narzędziami wyższego poziomu. W przeciwieństwie do języków takich jak C Python posiada cały szereg narzędzi i mechanizmów do zaawansowanego przetwarzania łańcuchów tekstu, a jednocześnie nie ma specjalnego typu dla pojedynczego znaku (jak `char` z C); zamiast tego tworzymy po prostu łańcuchy jednoznakowe.

Ścisłe mówiąc, łańcuchy znaków Pythona zaliczane są do *sekwencji niemutowalnych*, co oznacza, że ich zawartość jest uporządkowana od lewej strony do prawej, a samych łańcuchów nie można modyfikować w miejscu. Tak naprawdę łańcuchy znaków są pierwszym

reprezentantem większej klasy obiektów zwanych *sekwencjami*. Warto zwrócić szczególną uwagę na omawiane w niniejszym rozdziale operacje na sekwencjach, ponieważ będą one działały w ten sam sposób na innych typach sekwencji, którymi zajmiemy się później — na przykład na listach czy krotkach.

W tabeli 7.1 zamieszczono przegląd popularnych literałów i operacji na łańcuchach znaków, które zostaną przedstawione w niniejszym rozdziale. Puste łańcuchy znaków zapisuje się jako parę cudzysłówów lub apostrofów bez żadnych znaków między nimi. Istnieje wiele sposobów kodowania łańcuchów znaków. Jeżeli chodzi o przetwarzanie, łańcuchy znaków obsługują *wyrażenia*, takie jak konkatenacja (łączenie łańcuchów), wycinki (ekstrakcja części łańcuchów), indeksowanie (pobieranie elementów zgodnie z wartością przesunięcia) i wiele innych. Poza wyrażeniami Python udostępnia również zbiór *metod* łańcuchów znaków, które implementują często spotykane zadania związane z tym typem danych, a także *moduły* służące do bardziej zaawansowanego przetwarzania tekstu, w tym na przykład dopasowywania wzorców. Wszystkie te operacje omówimy w dalszej części rozdziału.

Tabela 7.1. Popularne literały i operacje na łańcuchach znaków

Operacja	Interpretacja
S = ''	Pusty łańcuch znaków
S = "mielonka o nazwie 'Mielonka'"	Cudzysłowy
S = 'm\ni\te\x00lonka'	Sekwencje znaków specjalnych
S = """...wiele wierszy..."""	Blok w potrójnych cudzysłowych
S = r'\temp\mielonka'	Surowy łańcuch znaków (bez znaków ucieczki)
S = b'mielonka'	Bajtowe łańcuchy znaków; w wersji 2.6, 2.7 i 3.x (rozdziały 4. i 37.)
S = u'mielonka'	Łańcuch znaków Unicode; w wersji 2.x i 3.3+ (rozdziały 4. i 37.)
S1 + S2 S * 3	Konkatenacja, powtóżenie
S[i] S[i:j] len(S)	Indeksowanie, wycinek, długość
"mała %s papuga" % kind	Wyrażenie formatujące łańcuch znaków
"a {0} papuga".format(kind)	Formatowanie łańcuchów znaków w 2.6, 2.7 i 3.x
S.find('ie')	Wywołania metod łańcuchów znaków (wszystkie 43 metody znajdziesz w dalszej części rozdziału): wyszukiwanie
S.rstrip()	Usuwanie białych znaków
S.replace('ie', 'xx')	Zastępowanie
S.split(',')	Dzielenie w miejscu wystąpienia separatora

S.isdigit()	Sprawdzania zawartości
S.lower()	Konwersja wielkości liter
S.endswith('mielonka')	Sprawdzenie końcówki łańcucha znaków
'mielonka'.join(strlist)	Złączenie z użyciem separatora
S.encode('latin-1')	Kodowanie Unicode
B.decode('utf8')	Dekodowanie Unicode itd. (patrz tabela 7.3)
<pre>for x in S: print(x) 'mielonka' in S [c * 2 for c in S] map(ord, S)</pre>	Iteracja, przynależność
re.match('mi(.*)nka', line)	Dopasowywanie wzorca; moduł biblioteki

Poza podstawowym zestawem narzędzi przeznaczonych do pracy z łańcuchami znaków (zobacz tabelę 7.1) Python obsługuje również bardziej zaawansowane przetwarzanie łańcuchów znaków w oparciu o dopasowanie wzorców dzięki wprowadzonemu w rozdziale 4. modułowi `re` (od ang. *regular expressions* — wyrażenia regularne) z biblioteki standardowej, a nawet wysokopoziomowe narzędzia, jak parsery XML, omówione pokrótko w rozdziale 37. W tej książce zajmiemy się jednak zagadnieniami podstawowymi, wymienionymi w tabeli 7.1.

Na początku rozdziału zamieszczałyśmy omówienie podstaw, czyli przegląd form literałów i wyrażeń tekstowych, następnie przejdzieliśmy do bardziej zaawansowanych narzędzi, jak metody łańcuchów znaków i formatowanie. Python oferuje wiele narzędzi służących do pracy z łańcuchami znaków. Nie będziemy tu omawiać wszystkich z nich, kompletną dokumentację można znaleźć w podręczniku biblioteki standardowej Pythona. Naszym celem jest omówienie najpowszechniej stosowanych narzędzi i pokazanie reprezentatywnych przykładów. Metody, których nie zademonstrujemy, mają w większości zastosowanie analogiczne do tych omówionych.

Literały łańcuchów znaków

Generalnie łańcuchy znaków są w Pythonie łatwe w użyciu. Chyba najbardziej skomplikowanym aspektem jest to, że istnieje tyle sposobów zapisania ich w kodzie.

- apostrofy — 'mie"lonka'
- cudzysłowy — "mie'lonka"
- potrójne apostrofy lub cudzysłowy — '"""...mielonka....", """...mielonka...."""
- sekwencje ze znakami ucieczki — "mi\tel\no\0nka"
- surowe łańcuchy znaków — r"C:\nowe\test.spm"
- literały bajtowe w 3.x i 2.6+ (zobacz rozdziały 4. i 37.): b'miel\x01lonka'
- literały Unicode w 2.x i 3.3+ (zobacz rozdziały 4. i 37.): u'jajka\u0020mielonka'

Najczęściej spotykane są bez wątpienia formy z apostrofami i cudzysłowami. Pozostałe pełnią wyspecjalizowane role, a omówienie ostatnich dwóch odłożymy do rozdziału 37. Przyjrzyjmy się krótko każdej z powyższych opcji.

Łańcuchy znaków w apostrofach i cudzysłowach są tym samym

W przypadku łańcuchów znaków Pythona znaki apostrofów i cudzysłówów mogą być stosowane wymiennie. Oznacza to, że literały łańcuchowe można zapisać zarówno w dwóch cudzysłowach, jak i dwóch apostrofach — obie te formy działają w ten sam sposób i zwracają ten sam typ obiektu. Dwa poniższe łańcuchy znaków będą na przykład identyczne:

```
>>> 'żywopłot', "żywopłot"  
('żywopłot', 'żywopłot')
```

Powodem używania dwóch form jest możliwość osadzenia wewnątrz łańcucha znaków znaku cudzysłowa (w łańcuchu ograniczonym apostrofami) i odwrotnie, bez konieczności stosowania sekwencji ze znakiem ucieczki (lewy ukośnik). Apostrof można osadzić w łańcuchu znaków w cudzysłowach, natomiast cudzysłów — w łańcuchu w apostrofach.

```
>>> 'ryce"rz', "ryce'rz"  
('ryce"rz', "ryce'rz")
```

W tej księdze ogólnie wolimy ujmować ciągi znaków w *apostrofy* tylko dlatego, że takie ciągi są nieznacznie łatwiejsze do odczytania, z wyjątkiem przypadków, w których w ciągu znaków jest osadzony apostrof. Jest to czysto subiektywny wybór stylu, ale Python wyświetla również ciągi w taki sposób, a większość programistów Pythona robi dokładnie to samo, więc prawdopodobnie również powinieneś.

Pamiętaj, że przecinek jest tutaj ważny; bez niego Python *automatycznie dokonuje konkatenacji* przylegających do siebie literałów łańcuchowych w każdym wyrażeniu, choć równie proste jest jawne wywołanie konkatenacji poprzez wstawienie między te łańcuchy operatora + (jak zobaczymy w rozdziale 12., ujęcie tego wyrażenia w nawiasy pozwala zapisać je w kilku wierszach).

```
>>> title = "Żywot " 'Briana' # Niejawnna konkatenacja  
>>> title  
'Żywot Briana'
```

Warto zauważyć, że wstawienie między łańcuchy znaków przecinków zwróciłoby krotkę, a nie łańcuch znaków. Widać również, że we wszystkich danych wyjściowych Python woli wyświetlać łańcuchy znaków w apostrofach, o ile same nie zawierają one takiego znaku. Apostrofy i cudzysłowy można również umieszczać w łańcuchach ze znakami ucieczki.

```
>>> 'ryce\'rz', "ryce\"rz"  
("ryce'rz", 'ryce"rz')
```

Aby zrozumieć ich działanie, powinieneś dowiedzieć się, czym w ogóle są sekwencje ze znakami ucieczki.

Sekwencje ucieczki reprezentują znaki specjalne

W ostatnim przykładzie cudzysłów i apostrof osadzono w łańcuchu znaków, poprzedzając go znakiem lewego ukośnika (\). Jest to ogólny wzorzec stosowany w łańcuchach znaków — lewe ukośniki wykorzystywane są do wprowadzenia specjalnego kodowania bajtów zwanego *sekwencją ucieczki* (ang. *escape sequence*).

Sekwencje ucieczki pozwalają osadzać w łańcuchach znaki, których nie da się łatwo wpisać z klawiatury. Znak \ i jeden lub większa liczba następujących po nim znaków w literale

łańcuchowym w wynikowym obiekcie łańcucha znaków zastępowane są pojedynczym znakiem o wartości binarnej określonej przez sekwencję ucieczki. Na przykład poniżej przedstawiamy łańcuch składający się z pięciu znaków, w którym osadzony jest znak nowego wiersza i tabulator.

```
>>> s = 'a\nb\tc'
```

Sekwencja znaków `\n` reprezentuje jeden znak specjalny — binarną wartość znaku nowego wiersza w określonym zestawie znaków (w przypadku zestawu ASCII będzie to znak o kodzie 10). W podobny sposób sekwencja `\t` zastępowana jest pojedynczym znakiem tabulatora. Wygląd wyświetlanego łańcucha znaków zależy od sposobu jego wyświetlania. W sesji interaktywnej znaki specjalne zwracane są w postaci z sekwencjami ucieczki, natomiast instrukcja `print` wyświetla je po zinterpretowaniu.

```
>>> s  
'a\nb\tc'  
>>> print(s)  
a  
b      c
```

Aby upewnić się, ile bajtów naprawdę zawiera łańcuch znaków, należy skorzystać z wbudowanej funkcji `len`. Zwraca ona prawdziwą liczbę bajtów w łańcuchu znaków bez względu na sposób kodowania bądź wyświetlania tego łańcucha.

```
>>> len(s)  
5
```

Łańcuch ten składa się z pięciu znaków: pierwszy znak to `a` z zestawu ASCII, następnie mamy znak nowego wiersza, później znak `b` z zestawu ASCII i tak dalej.



Jeżeli jesteś przyzwyczajony do tekstów składających się w całości ze znaków ASCII, możesz pomyśleć, że oznacza to również, iż nasz łańcuch znaków zajmuje 5 bajtów. Nie powinieneś jednak tak myśleć. W praktyce w świecie Unicode „bajty” nie mają żadnego znaczenia. Po pierwsze obiekt łańcucha znaków naprawdopodobniej zajmuje w pamięci operacyjnej znacznie więcej miejsca.

Co ważniejsze, zarówno zawartość, jak i długość łańcucha znaków jest w świecie Unicode mocno powiązana z *kodami znaków* (to swego rodzaju numery identyfikacyjne znaków w danym zestawie), gdzie pojedynczy znak niekoniecznie musi odpowiadać jednemu bajtowi, zarówno gdy jest zapisany w pliku, jak i przechowywany w pamięci. Takie odwzorowanie jeden-do-jednego może być prawdziwe dla prostego 7-bitowego tekstu ASCII, ale nawet to zależy zarówno od zewnętrznego typu kodowania, jak i zastosowanego rozwiązania pamięci wewnętrznej. Na przykład w schemacie UTF-16 znaki ASCII mogą zajmować 1, 2 lub 4 bajty, w zależności od tego, jak Python przydziela im miejsce w pamięci. W przypadku tekstu innego niż ASCII, którego kody znaków mogą być zbyt duże, aby zmieściły się w 8-bitowym bajcie, odwzorowanie znaków na bajty w ogóle nie ma zastosowania.

Warto tutaj zauważyć, że w wersji 3.x Python definiuje ciągi znaków formalnie jako *sekwencje kodów znaków Unicode*, a nie bajtów. Więcej szczegółowych informacji na temat tego, jak ciągi znaków są przechowywane wewnętrznie w pamięci, znajdziesz w rozdziale 37. Aby na razie uniknąć kłopotów z łańcuchami znaków, na wszelki wypadek powinieneś o nich myśleć w kategoriach znaków, a nie bajtów. Zaufaj mi; jako były programista C również musiałem zerwać z tym nawykiem!

Zwróć uwagę, że oryginalne znaki lewego ukośnika, pokazane w poprzednim przykładzie, nie są przechowywane w pamięci wraz z łańcuchami znaków, a jedynie sygnalizują Pythonowi, że następujące po nich znaki ma traktować w specjalny sposób. W celu zakodowania tych specjalnych bajtów Python wykorzystuje pełny zbiór sekwencji kodów ucieczki, zaprezentowany w tabeli 7.2.

Tabela 7.2. Sekwencje ucieczki ze znakiem lewego ukośnika

Sekwencja ucieczki	Znaczenie
\<klawisz Enter>	Znak kontynuacji wiersza
\\"	Ukośnik lewy (zachowuje jeden znak \)
\'	Apostrof (zachowuje znak ')
\"	Cudzysłów (zachowuje znak ")
\a	Sygnal dźwiękowy
\b	Znak cofania (ang. <i>backspace</i>)
\f	Wysunięcie strony (ang. <i>form feed</i>)
\n	Nowy wiersz (wysunięcie wiersza, ang. <i>line feed</i>)
\r	Powrót karetki
\t	Tabulator poziomy
\v	Tabulator pionowy
\xhh	Znak w zapisie szesnastkowym hh (dokładnie dwie cyfry)
\ooo	Znak w zapisie ósemkowym ooo (najwyżej trzy cyfry)
\0	Null — binarny zapis znaku o kodzie 0 (nie sygnalizuje końca łańcucha znaków)
\N{ id }	id to nazwa znaku w bazie Unicode (np. w "\N{Latin Small Letter A with ogonek}", co odpowiada literze „ą”, zwraca uwagę użycie w nazwie swojsko brzmiącego „ogonka” — przyp. tłumacza)
\uhhhh	Szesnastobitowy kod znaku Unicode (zapisany w postaci dwóch liczb hex)
\Uhhhhhhhh	Trzydziestodwubitowy kod znaku Unicode (zapisany w postaci czterech liczb hex) ^[1]
\innny	Nie jest sekwencją ucieczki (zachowuje znak \ oraz znak następujący po nim)

Niektóre sekwencje ucieczki pozwalają na osadzanie bezwzględnych wartości binarnych w znakach łańcucha. Jako przykład poniżej zamieszczamy łańcuch składający się z pięciu znaków, w którym osadzono dwa binarne bajty zerowe (zakodowane jako sekwencje ucieczki z jednocyfrowymi wartościami w zapisie ósemkowym).

```
>>> s = 'a\0b\0c'
```

```
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

W Pythonie bajt zerowy (`null`) nie kończy łańcucha znaków w taki sposób, jak dzieje się to w języku C. Zamiast tego Python zachowuje zarówno długość łańcucha znaków, jak i jego tekst w pamięci. Tak naprawdę w Pythonie żaden znak nie kończy łańcucha znaków. Poniżej znajduje się łańcuch znaków składający się z samych bezwzględnych, binarnych kodów ucieczki — binarnych 1 i 2 (zakodowanych w postaci ósemkowej), po których znajduje się binarne 3 (zakodowane w postaci szesnastkowej).

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

Python wyświetla niedrukowalne znaki w postaci kodów ósemkowych, niezależnie od tego, w jaki sposób zostały zadeklarowane. W literałach można dowolnie mieszać sekwencje ucieczki z wartościami bezwzględnymi z symbolicznymi sekwencjami ucieczki, przedstawionymi w tabeli 7.2. Poniższy listing zawiera ciąg znaków `mielonka` przemieszany ze znakami tabulacji, znakiem nowego wiersza oraz binarnym znakiem o wartości zero, zapisanym w postaci szesnastkowej.

```
>>> S = "m\ti\n\x00lonka"
>>> S
'm\ti\n\x00lonka'
>>> len(S)
11
>>> print(S)
m      i
e lonka
```

Ma to dużo większe znaczenie, kiedy w Pythonie przetwarzamy pliki z danymi binarnymi. Ponieważ ich zawartość w skryptach reprezentowana jest jako łańcuchy znaków, możemy przetwarzać pliki binarne zawierające wszelkiego rodzaju wartości binarne — po otwarciu pliku w trybie binarnym będziemy odczytywać z niego ciągi znaków reprezentujących nieprzetworzone wartości bajtów (więcej szczegółowych informacji na temat plików znajdziesz w rozdziale 4., rozdziale 9. i rozdziale 37.).

Wreszcie, tak jak to pokazaliśmy w ostatnim wierszu tabeli 7.2, jeżeli Python nie rozpozna znaku umieszczonego po \ jako poprawnego kodu ucieczki, po prostu zapisuje znak ukośnika w wynikowym łańcuchu znaków.

```
>>> x = "C:\py\code"                                # Zachowuje dosłowny znak \
wyświetla go jako \\)
>>> x
'C:\\py\\code'
```

```
>>> len(x)
```

```
10
```

Jeżeli jednak nie jesteś w stanie zapamiętać całej tabeli 7.2 (i są prawdopodobnie lepsze zastosowania dla Twojej pamięci!), to raczej nie powinieneś polegać na takim zachowaniu. Aby jawnie użyć znaku lewego ukośnika w ciągu znaków, powinieneś użyć dwóch znaków lewego ukośnika (\ \ pełni rolę sekwencji ucieczki dla znaku \) lub użyj „surowych” ciągów znaków; w następnym podrozdziale dowiesz się, jak to zrobić.

Surowe łańcuchy znaków blokują sekwencje ucieczki

Jak widzieliśmy, sekwencje ucieczki przydają się do osadzania specjalnych kodów bajtów wewnętrz łańcuchów znaków. Czasami jednak specjalne traktowanie ukośników lewych służących jako wprowadzenie do sekwencji ucieczki może powodować problemy. Zaskakującą często można spotkać początkujących użytkowników Pythona, którzy próbują otworzyć plik, podając jego nazwę jako argument w następującej postaci:

```
myfile = open('C:\nowy\tekst.dat', 'w')
```

i myślą, że w ten sposób uda im się otworzyć plik o nazwie *tekst.dat* umieszczoney w katalogu *C:\nowy*. Problemem jest jednak to, że \n jest w takiej sytuacji interpretowany jako znak nowego wiersza, a sekwencja \t zastępowana jest przez tabulator. W rezultacie takie wywołanie próbuje otworzyć plik o nazwie *C:<nowy wiersz>owy<tabulator>ekst.dat*, a wynik takiej operacji z reguły okazuje się dość daleki od oczekiwania.

W takiej właśnie sytuacji przydają się surowe łańcuchy znaków (ang. *raw strings*). Jeżeli litera r (mała lub wielka) pojawia się tuż przed cudzysłowem lub apostrofem otwierającym łańcuch znaków, wyłącza ona mechanizm sekwencji ucieczki. W rezultacie Python zachowuje dosłowne znaki ukośników dokładnie tak, jak je wpisaliśmy. Z tego powodu, aby naprawić problem z nazwą pliku, wystarczy pamiętać o poprzedzeniu ciągu znaków literą r:

```
myfile = open(r'C:\nowy\tekst.dat', 'w')
```

Ponieważ jednak dwa znaki lewego ukośnika są sekwencją ucieczki dla pojedynczego znaku \, ten sam efekt możemy osiągnąć w następujący sposób:

```
myfile = open('C:\\nowy\\\\tekst.dat', 'w')
```

Tak naprawdę Python również wykorzystuje czasami podwajanie znaków lewego ukośnika przy wyświetlanie ciągów zawierających znak \.

```
>>> path = r'C:\nowy\tekst.dat'  
>>> path # Wyświetl jako kod Pythona  
'C:\\nowy\\\\tekst.dat'  
>>> print(path) # Format przyjazny dla  
użytkownika  
C:\\nowy\\tekst.dat  
>>> len(path) # Długość łańcucha znaków
```

17

Podobnie jak w przypadku reprezentacji liczbowej, wyniki w sesji interaktywnej domyślnie wyświetlane są w taki sposób, jakby były one kodem, stąd w wynikach działania znajdują się sekwencje ucieczki dla lewych ukośników. Instrukcja `print` udostępnia format bardziej przyjazny dla użytkowników, pokazujący, że tak naprawdę w każdym miejscu znajduje się tylko jeden znak ukośnika. Aby to zweryfikować, możemy sprawdzić wynik działania wbudowanej

funkcji `len`, która zwraca liczbę znaków łańcucha niezależnie od formatu jego wyświetlania. Jeżeli policzysz znaki w wynikach działania funkcji `print(path)`, przekonasz się, że każda sekwencja `\\"` jest traktowana jako pojedynczy znak `\`, co razem daje 17 znaków dla całego łańcucha.

Poza ścieżkami do katalogów w systemie Windows surowe łańcuchy znaków wykorzystywane są często w wyrażenях regularnych (dopasowywaniu wzorców obsługiwanych przez moduł `re` przedstawiony w rozdziałach 4. i 37.). Warto również zauważyć, że w skryptach Pythona ścieżki do katalogów w systemach Windows i Unix wykorzystują zazwyczaj znaki *prawych ukośników* — dzieje się tak, ponieważ Python próbuje interpretować ścieżki w sposób przenośny (to znaczy, że w systemie Windows możemy otworzyć plik, używając ścieżki zapisanej jako np. `'C:/nowy/tekst.dat'`). Surowe łańcuchy znaków przydają się jednak w sytuacjach, kiedy zapisujesz ścieżki w systemie Windows w tradycyjny sposób, przy użyciu znaków lewego ukośnika.



Warto zauważyć, że nawet surowy łańcuch znaków nie może się kończyć lewym ukośnikiem, ponieważ taki pojedynczy ukośnik zostanie potraktowany jako znak specjalny zmieniający znaczenie znaku cudzysłowu domykającego dany literał, co spowoduje błąd składni. Innymi słowy, łańcuch znaków `r"..."\\"` nie jest prawidłowym literałem — surowy ciąg znaków nie może się kończyć nieparzystą liczbą znaków lewego ukośnika. Jeżeli chcesz zakończyć surowy łańcuch znakiem ukośnika, powinieneś użyć dwóch ukośników, a następnie zastosować operator wycinania, pozbywając się zbędnego znaku (`r'1\nb\tc\\'[:-1]`), albo dokleić ten znak (`r'1\nb\tc' + '\\'`) lub ewentualnie zrezygnować z surowych łańcuchów znaków i użyć zwykłego łańcucha znaków ze zdublowanymi znakami lewego ukośnika (`'1\\nb\\tc\\'`). Wszystkie trzy powyższe techniki dają w efekcie ten sam, ośmioznakowy łańcuch znaków zawierający trzy lewe ukośniki.

Potrójne cudzysłowy i apostrofy kodują łańcuchy znaków będące wielowierszowymi blokami

Jak na razie zapoznaliśmy się z apostrofami, cudzysłowami, sekwencjami ucieczki i surowymi łańcuchami znaków. Python posiada również format literału łańcucha znaków z potrójnymi apostrofami lub cudzysłowami, czasem nazywany *blokowym łańcuchem znaków* (ang. *block string*), który jest składniowym ułatwieniem służącym do zapisu wielowierszowych danych tekstowych. Format ten rozpoczyna się od trzech apostrofów lub cudzysłów, potem następuje dowolna liczba wierszy tekstu, a na końcu sekwencja trzech apostrofów lub cudzysłów — w zależności od tego, jaka sekwencja otwierała łańcuch. Cudzysłowy i apostrofy osadzone w tekście łańcucha znaków mogą być zastępowane sekwencjami ucieczki, ale nie muszą — łańcuch ten nie kończy się, dopóki Python nie napotka sekwencji trzech znaków, które zostały użyte do otwarcia łańcucha, tak jak to zostało pokazane w przykładzie poniżej (znaki `...` oznaczają tutaj podpowiedź powłoki Pythona wskazującą na kontynuację wiersza; są wyświetlane w środowiskach innych niż IDLE; nie powinieneś ich wpisywać samodzielnie).

```
>>> mantra = """Zawsze patrz
... na jasną
... stronę życia."""
>>>
>>> mantra
'Zawsze patrz\n na jasną\nstronę życia.'
```

Ten łańcuch znaków rozciąga się na trzy wiersze (w niektórych interfejsach interaktywny znak zachęty zmienia się na ... w wierszach będących kontynuacją; w środowisku IDLE po prostu przechodzi do wiersza niżej). Python zbiera cały tekst mieszący się w potrójnych cudzysłowach lub apostrofach do jednego, wielowierszowego łańcucha znaków ze znakami nowego wiersza (\n) osadzonymi w miejscach, gdzie w kodzie znajduje się złamanie wiersza. Warto również zauważyc, że w drugim wierszu literał na początku znajduje się spacja, jednak w trzecim wierszu już jej nie ma — otrzymujemy dokładnie to, co wpisaliśmy. Gdy chcesz zobaczyć nasz łańcuch znaków z właściwie zinterpretowanymi znakami końca wierszy, wtedy zamiast bezpośrednio wyświetlać wartość zmiennej, powinieneś użyć funkcji print.

```
>>> print(mantra)
```

Zawsze patrz

na jasną

stronę życia.

W rzeczywistości łańcuchy z potrójnymi cudzysłowami zachowają cały załączony tekst, włącznie z tekstami umieszczonymi po prawej stronie kodu, które w zamierzeniu miały być komentarzami. Więc nie rób tego — umieść swoje komentarze powyżej lub poniżej cytowanego tekstu lub użyj automatycznego łączenia ciągów znaków (wspominaliśmy o tym wcześniej), z wyraźnie oznaczonymi znakami nowego wiersza (jeżeli to konieczne), oraz otaczającymi nawiasami, co pozwoli na odpowiednie sformatowanie tekstu; więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 10. i rozdziale 12., w których będziemy omawiać reguły składni:

```
>>> menu = """szynka # komentarze umieszczone tutaj zostaną dodane do tekstu!
... jajka           # o tym mówimy
...
>>> menu
'szynka # komentarze umieszczone tutaj zostaną dodane do tekstu!\njajka # o
tym mówimy\n'
>>> menu =
... "szynka\n" # tutaj komentarze są ignorowane
... "jajka\n" # ale nowe wiersze nie są tworzone automatycznie
...
>>> menu
'szynka\njajka\n'
```

Łańcuchy znaków tego rodzaju przydają się wtedy, gdy w programie potrzebny jest tekst wielowierszowy — na przykład, kiedy chcemy w plikach z kodem źródłowym osadzić wielowierszowe komunikaty o błędach lub kod HTML, XML czy JSON. Takie bloki można osadzać bezpośrednio w skryptach bez konieczności korzystania z zewnętrznych plików tekstowych czy jawniej konkatenacji i znaków nowych wierszy.

Łańcuchy znaków z potrójnymi cudzysłowami lub apostrofami są również często wykorzystywane przy dokumentowaniu kodu, czyli literałach łańcuchowych traktowanych jako komentarze, które pojawiają się w różnych miejscach programu (więcej informacji na ten temat później). Oczywiście nie muszą one być blokami z potrójnymi cudzysłowami czy apostrofami, ale najczęściej tak robimy, kiedy chcemy w nich było umieszczać komentarze wielowierszowe.

Wreszcie łańcuchy znaków tego typu są często wykorzystywane jako paskudny sposób *tymczasowego wyłączania* wierszy kodu w trakcie programowania (no dobrze, nie tak całkiem paskudny i w dodatku powszechnie stosowany, ale nie o to mi tutaj chodziło). Krótko mówiąc, jeżeli chcesz wyłączyć kilka wierszy kodu i ponownie wykonać skrypt, wystarczy przed tym kodem i po nim wstawić trzy apostrofy lub cudzysłowy.

```
X = 1
"""

import os                  # Tymczasowo blokujemy wykonanie tego kodu
print(os.getcwd())
"""

Y = 2
```

Nazwałem ten sposób paskudnym, ponieważ Python naprawdę tworzy łańcuch znaków z wierszy wyłączonych w ten sposób; nie ma to jednak większego wpływu na wydajność programu. W przypadku większych fragmentów kodu jest to również łatwiejsze wyjście od ręcznego wstawiania znaków # na początku każdego wiersza, a później usuwania ich. Jest to szczególnie przydatne, kiedy używamy edytora tekstu, który nie obsługuje edycji kodu napisanego w Pythonie w specjalny sposób. W Pythonie praktyczność często wygrywa z estetyką.

Łańcuchy znaków w akcji

Po utworzeniu łańcuchów znaków za pomocą omówionych wyrażeń z literałami z pewnością będziemy chcieli coś z nimi zrobić. W tym podrozdziale i kolejnych dwóch zademonstrujemy wyrażenia, metody i formatowanie łańcuchów znaków — najważniejsze narzędzia służące w Pythonie do przetwarzania tekstu.

Podstawowe operacje

Rozpoczniemy od zaangażowania interpretera Pythona w sesji interaktywnej w celu zaprezentowania podstawowych operacji na łańcuchach znaków, wymienionych w tabeli 7.1. Łańcuchy znaków można ze sobą łączyć za pomocą operatora konkatenacji +, a także powtarzać za pomocą operatora *.

```
% python
>>> len('abc')                               # Długość (len), czyli liczba
znaków
3
>>> 'abc' + 'def'                           # Konkatenacja – nowy łańcuch
znaków
'abcdef'
>>> 'Ni!' * 4                                # Powtórzenie – jak "Ni!" +
"Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Wbudowana funkcja `len` zwraca długość łańcucha (lub dowolnego innego obiektu, który ma jakąś długość). Z formalnego punktu widzenia dodanie do siebie dwóch obiektów łańcuchów znaków za pomocą operatora `+` tworzy nowy obiekt łańcucha znaków, który łączy zawartość obu argumentów. Powtórzenie za pomocą operatora `*` przypomina dodawanie łańcucha znaków do samego siebie kilka razy. W obu przypadkach Python pozwala na utworzenie łańcuchów znaków o dowolnym rozmiarze. W Pythonie nie trzeba niczego wcześniej deklarować, dotyczy to również rozmiarów struktur danych — po prostu obiekty łańcuchowe tworzysz na bieżąco w miarę potrzeb i pozwalasz Pythonowi automatycznie zarządzać alokacją pamięci (więcej informacji na temat mechanizmu zarządzania pamięcią znajdziesz w rozdziale 6.).

Powtórzenie może się na początku wydawać nieco dziwne, ale przydaje się w zaskakująco dużej liczbie zastosowań. Na przykład, aby wyświetlić wiersz składający się z osiemdziesięciu myślników, można albo odliczyć osiemdziesiąt takich znaków, albo pozwolić, aby to Python je za nas policzył.

```
>>> print('----- ...więcej... ---')      # 80 myślników, gorszy sposób  
>>> print('-'*80)                      # 80 myślników, lepszy sposób
```

Warto zauważyć, że działa tutaj przeciążanie operatorów — te same operatory `+` i `*` w przypadku liczb wykonują dodawanie i mnożenie. Python wykonuje te operacje poprawnie, ponieważ zna typy dodawanych i mnożonych obiektów. Należy jednak pamiętać — reguły nie są aż tak liberalne, jak można by oczekiwać. Python nie pozwala na przykład łączyć liczb i łańcuchów znaków w wyrażeniach z operatorem `+`, zatem wyrażenie `'abc' + 9` zwróci błąd, zamiast automatycznie przekonwertować liczbę 9 na łańcuch znaków.

Jak pokazano w pobliżu końca tabeli 7.1, możemy również wykonywać iteracje na łańcuchach znaków za pomocą pętli `for`, a także sprawdzać przynależność znaków lub podłańcuchów za pomocą wyrażenia z operatorem `in` (co jest tak naprawdę formą wyszukiwania). W przypadku podłańcuchów operator `in` działa podobnie do metody `str.find()` omówionej w dalszej części tego rozdziału, ale zwraca wartość logiczną typu Boolean zamiast pozycji podłańcucha w źródłowym łańcuchu znaków (w poniższym przykładzie użyto wywołania funkcji `print` w wersji 3.x, stąd w wyświetlonym wierszu mogą pojawić się niewielkie wcięcia; w wersji 2.x zamiast tego powinieneś użyć polecenia `print c`):

```
>>> myjob = "haker"  
>>> for c in myjob: print(c, end=' ')      # Przechodzenie przez kolejne  
elementy i wyświetlanie ich (dla wersji 3.x)  
  
...  
h a k e r  
>>> "k" in myjob                         # Znaleziono  
True  
>>> "z" in myjob                          # Nie znaleziono  
False  
>>> 'jajko' in 'abcdejajkoxyz'           # Wyszukiwanie podcięgu znaków,  
pozycja nie jest zwracana  
True
```

Pętla `for` przypisuje zmienną do kolejnych elementów sekwencji (tutaj: łańcucha znaków) i wykonuje dla każdego z nich jedno lub więcej poleceń. W rezultacie zmienna `c` staje się kursorem przechodzącym przez kolejne elementy łańcucha znaków. Więcej szczegółowych informacji na temat narzędzi iteracyjnych omówionych w tabeli 7.1 i innych znajdziesz w dalszej części książki (zwłaszcza w rozdziałach 14. i 20.).

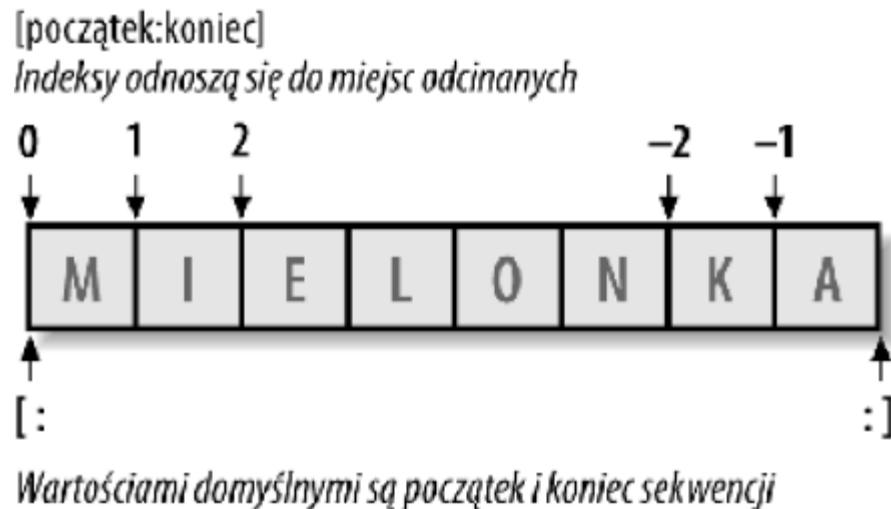
Indeksowanie i wycinki

Ponieważ łańcuchy znaków definiowane są jako uporządkowane kolekcje znaków, dostęp do poszczególnych elementów można uzyskać według ich pozycji. W Pythonie znaki z ciągu są pobierane poprzez *indeksowanie*, gdzie po nazwie ciągu podajemy w nawiasach kwadratowych numeryczną wartość przesunięcia żądanego komponentu i w efekcie otrzymujemy łańcuch zawierający jeden znak znajdujący się na podanej pozycji.

Tak jak w języku C, również w Pythonie wartości przesunięcia w ciągach znaków są liczone od 0 i kończą się na wartości o jeden mniejszej od długości łańcucha. W przeciwieństwie do C Python pozwala jednak pobierać elementy z sekwencji również za pomocą *ujemnych* wartości przesunięcia. Z technicznego punktu widzenia ujemna wartość przesunięcia dodawana jest do długości łańcucha znaków w celu uzyskania pożądanej wartości dodatniej. Ujemne wartości przesunięcia można sobie również wyobrazić jako odliczanie pozycji znaku od końca łańcucha. Widać to w przykładzie poniżej.

```
>>> S = 'mielonka'  
>>> S[0], S[-2] # Indeksowanie od początku lub  
od końca  
(‘m’, ‘k’)  
>>> S[1:3], S[1:], S[::-1] # Wycinek – ekstrakcja części  
łańcucha  
(‘ie’, ‘ielonka’, ‘mielonk’)
```

Pierwszy wiersz definiuje łańcuch składający się z ośmiu znaków i przypisuje do niego zmienną S. W kolejnym wierszu łańcuch ten zostaje zindeksowany na dwa sposoby. S[0] pobiera element znajdujący się na pozycji o wartości przesunięcia równej 0 (jednoznakowy łańcuch znaków ‘m’). S[-2] pobiera element znajdujący się na pozycji o wartości przesunięcia 2 liczonej od końca (lub, odpowiednio, 8 + (-2) od początku). Wartości przesunięcia i wycinki można przedstawić np. w postaci komórek, jak pokazano na rysunku 7.1 [2].



Rysunek 7.1. Wartości przesunięcia i wycinki. Dodatnie wartości przesunięcia liczone są od lewej strony (pierwszy element ma wartość 0), natomiast wartości ujemne — od prawej (ostatni element ma wartość przesunięcia równą -1). W indeksowaniu i tworzeniu wycinków można korzystać z obu sposobów określania wartości przesunięcia

Ostatni wiersz powyższego przykładu demonstruje tworzenie tzw. *wycinków* (ang. *slice*). Stanowią one uogólnioną formę indeksowania, które zwraca całą *sekcję*, a nie pojedynczy element. Tworzenie wycinków najlepiej jest sobie wyobrazić jako formę analizy składniowej (*parsowania*), w szczególności przy zastosowaniu do łańcuchów znaków. Pozwala to na ekstrakcję całej sekcji, lub inaczej mówiąc, całego podłańcucha znaków za jednym razem. Wycinki można wykorzystać do ekstrakcji kolumn danych czy odcinania tekstu znajdującego się na początku i końcu. W dalszej części rozdziału zobaczymy kolejny przykład tworzenia wycinków jako metodę analizy składniowej tekstu.

A oto jak działają wycinki. Kiedy indeksujemy obiekt sekwencji taki jak łańcuch znaków za pomocą pary wartości offsetów (przesunięć) rozdzielonych dwukropkiem, Python zwraca nowy obiekt zawierający całą sekcję (podciąg) znaków identyfikowaną przez tę parę wartości. Przedział taki jest lewostronnie zamknięty i prawostronnie otwarty, co oznacza, że Python pobiera wszystkie elementy od lewej wartości przesunięcia (*włącznie z nią*) do prawej wartości przesunięcia (*ale już bez niej samej*) i zwraca nowy obiekt zawierający pobrane elementy. Jeżeli którykolwiek z offsetów zostanie pominięty, jego wartością domyślną będzie odpowiednio zero i długość ciętego obiektu.

W przykładzie wyżej wyrażenie `S[1:3]` powodowało ekstrakcję elementów zajmujących pozycje o wartościach przesunięcia 1 i 2. Pobrane zostały zatem drugi i trzeci element, ale już nie element czwarty, mający wartość offsetu równą 3. Kolejny wycinek — `S[1:]` — pobiera *wszystkie elementy poza pierwszym*. Górną granicą, która nie jest podana, ma wartość domyślną równą długości łańcucha znaków. Wreszcie wyrażenie `S[:-1]` pobiera *wszystkie elementy z wyjątkiem ostatniego*, ponieważ granica dolna ma wartość domyślną równą 0, natomiast -1 odnosi się do ostatniego elementu i zarazem nie obejmuje go (przedział jest prawostronnie otwarty).

Może się to na pierwszy rzut oka wydawać mylące, jednak indeksowanie i wycinki są narzędziami prostymi w użyciu (kiedy nabierzesz już wprawy) i o dużych możliwościach. Pamiętaj jednak, że jeśli nie jesteś pewny, co oznacza dany wycinek, zawsze powinieneś go najpierw przetestować w sesji interaktywnej. W kolejnym rozdziale zobaczymy, że można nawet zmienić całą część określonego obiektu za jednym razem, przypisując ją do wycinka (ta technika nie działa jednak z wartościami niemutowalnymi, jak łańcuchy znaków). Poniżej znajduje się podsumowanie zagadnień związanych z indeksowaniem i wycinkami.

Indeksowanie (`S[i]`) pobiera komponenty znajdujące się na pozycji określonej wartością przesunięcia (offsetem).

- Pierwszy element ma wartość przesunięcia równą 0.
- Indeks ujemny oznacza odliczanie od końca (inaczej — od prawej strony).
- `S[0]` pobiera pierwszy element.
- `S[-2]` pobiera drugi element od końca (podobnie jak `S[len(S)-2]`).

Wycinek (`S[i:j]`) dokonuje ekstrakcji ciągłego fragmentu sekwencji.

- Górną granicą jest otwarta.
- Pominione granice wycinka mają wartości domyślne ustawione odpowiednio na 0 (dolina granica) i długość sekwencji (górna granica).
- `S[1:3]` pobiera elementy znajdujące się na pozycjach o wartościach przesunięcia od 1 do 2 włącznie (bez elementu 3).
- `S[1:]` pobiera elementy znajdujące się na pozycjach o wartościach przesunięcia od 1 do końca (długości) sekwencji.
- `S[:3]` pobiera elementy znajdujące się na pozycjach o wartościach przesunięcia od 0 do 2 włącznie (bez 3).
- `S[:-1]` pobiera elementy znajdujące się na pozycjach od wartości przesunięcia 0 do ostatniego elementu, ale bez niego.
- `S[:]` pobiera elementy znajdujące się na pozycjach od wartości przesunięcia 0 do końca — w rezultacie otrzymujemy pełną kopię łańcucha `S`.

Rozszerzone wycinki ($S[i:j:k]$) pozwalają na określenie kroku k , którego wartość domyślnie wynosi +1:

- Zdefiniowanie kroku pozwala na pomijanie wybranych elementów sekwencji i odwracanie ich kolejności — więcej szczegółowych informacji na ten temat znajdziesz w kolejnym podrozdziale.

Przedostatni element powyższego zestawienia okazuje się być często stosowaną sztuczką. Pozwala on wykonać pełną kopię obiektu sekwencji, czyli uzyskać kolejny obiekt o tej samej wartości, ale zajmujący odrębne miejsce w pamięci (więcej informacji na temat kopii obiektów możesz znaleźć w rozdziale 9.). Nie jest to zbyt przydatne w przypadku obiektów niemutowalnych, jakimi są łańcuchy znaków, jednak przydaje się w połączeniu z obiektami, które można modyfikować w miejscu, jak listy.

W kolejnym rozdziale zobaczymy również, że składnia wykorzystywana do indeksowania po offsetach (nawiasy kwadratowe) używana jest również do indeksowania słowników według kluczy; choć operacje te wyglądają tak samo, mają jednak różne znaczenie.

Rozszerzone wycinki — trzeci limit i obiekty wycinków

W Pythonie 2.3 i późniejszych wersjach wyrażenia z wycinkami otrzymały opcjonalny trzeci indeks wykorzystywany jako *krok* (w języku angielskim znany jako *step* lub *stride*). Krok dodawany jest do indeksu każdego pobieranego elementu. Pełną formą wycinka jest teraz $X[I:J:K]$, co oznacza, że należy dokonać ekstrakcji wszystkich elementów z ciągu X znajdujących się na pozycjach o wartościach przesunięcia od I do $J-1$, z krokiem K . Trzeci argument, K , ma wartość domyślną równą +1, dlatego normalnie pobierane są wszystkie elementy wycinka od lewej do prawej strony. Jeżeli jednak podamy inną wartość tego argumentu, trzecią wartość można wykorzystać do pomijania niektórych elementów sekwencji lub odwrócenia ich kolejności.

Na przykład wyrażenie $X[1:10:2]$ spowoduje pobranie *co drugiego* elementu z sekwencji X pomiędzy pozycjami przesunięcia o wartościach 1 do 9, co da elementy o offsetach 1, 3, 5, 7 oraz 9. Tak jak wcześniej, pierwszy i drugi argument mają wartości domyślne równe odpowiednio 0 i długości sekwencji, dzięki czemu wyrażenie $X[::-2]$ spowoduje wybranie *co drugiego* elementu od początku do końca sekwencji.

```
>>> S = 'abcdefghijklmno'  
>>> S[1:10:2]           # Pomijanie co drugiego elementu  
'bdfhj'  
>>> S[::-2]  
'acegikmo'
```

Krok może również być wartością ujemną, co pozwala na zbieranie elementów w odwrotnej kolejności. Na przykład wyrażenie "halo"[::-1] zwraca nowy łańcuch znaków "olah". Pierwsze dwie granice mają wartości domyślne odpowiednio 0 i długość sekwencji, natomiast krok o wartości -1 wskazuje na to, że wycinanie elementów powinno się odbywać od prawej strony do lewej zamiast — jak zazwyczaj — od lewej do prawej. W rezultacie cała sekwencja zostaje *odwrócona*.

```
>>> S = 'halo'  
>>> S[::-1]           # Odwracanie kolejności elementów  
'olah'
```

Po podaniu ujemnej wartości kroku znaczenie dwóch pierwszych granic właściwie się odwraca. Oznacza to, że wycinek $S[5:1:-1]$ powoduje pobranie elementów z pozycji przesunięcia od 2 do 5 w odwrotnej kolejności (wynik zawiera elementy z pozycji przesunięcia 5, 4, 3 oraz 2).

```
>>> S = 'abcdefg'  
>>> S[5:1:-1] # Zmiana znaczenia granic  
'fdec'
```

Podobne przeskakiwanie elementów i ich odwracanie to najczęstsze zastosowanie wycinków z trzema argumentami. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona; warto również spróbować na własną rękę wykonać kilka eksperymentów w sesji interaktywnej. Do wycinków tego typu powrócimy ponownie w dalszej części książki przy okazji omawiania instrukcji pętli `for`.

W dalszej części książki dowiemy się, że tworzenie wycinków jest w rzeczywistości operacją indeksowania z zastosowaniem *obiektu wycinka* zamiast liczby całkowitej określającej indeks. Jest to obserwacja kluczowa dla programistów tworzących własne klasy, które mają obsługiwać operacje indeksowania i wycinania.

```
>>> 'mielonka'[1:3] # Składnia wycinania  
'ie'  
>>> 'mielonka'[slice(1, 3)] # Obiekty wycinków  
'ie'  
>>> 'mielonka'[::-1]  
'aknoleim'  
>>> 'mielonka'[slice(None, None, -1)]  
'aknoleim'
```

Znaczenie wycinków

W całej książce będziemy zamieszczać ramki z praktycznymi przypadkami zastosowania omawianych rozwiązań (jak tutaj), które pozwolą zobaczyć, w jaki sposób niektóre możliwości języka są zazwyczaj używane w prawdziwych programach. Ponieważ trudno zrozumieć rzeczywiste przypadki zastosowania bez znajomości samego Pythona, ramki te będą zazwyczaj zawierały odniesienia do zagadnień, które jeszcze nie były wprowadzone. Można je również potraktować jako zapowiedź tego, w jaki sposób pewne abstrakcyjne koncepcje z języka przydają się w często wykonywanych zadaniach programistycznych.

Nieco później pokażemy na przykład, że argumenty wywołania podane w systemowym wierszu poleceń i wykorzystywane do uruchamiania programu w Pythonie są dostępne za pośrednictwem atrybutu `argv` wbudowanego modułu `sys`.

```
# Plik echo.py  
import sys  
print(sys.argv)  
% python echo.py -a -b -c  
['echo.py', '-a', '-b', '-c']
```

Zazwyczaj interesują nas tylko argumenty następujące po nazwie programu. I tutaj znajdujemy bardzo typowe zastosowanie wycinków — jedno wyrażenie z wycinkiem można wykorzystać do zwracania wszystkich elementów listy z wyjątkiem pierwszego. W tym przypadku wyrażenie `sys.argv[1:]` zwraca pożądaną listę argumentów wywołania `['-a', '-b', '-c']`. Możemy ją następnie przetwarzać bez konieczności zajmowania się nazwą programu, która znajdowała się na początku.

Wycinki są również często wykorzystywane do oczyszczania wierszy odczytywanych z plików na dysku. Jeżeli wiesz, że wiersz będzie się kończył znakiem końca wiersza (a właściwie znakiem \n oznaczającym nowy wiersz), możesz się go pozbyć za pomocą jednego wyrażenia, takiego jak `line[:-1]`, które powoduje ekstrakcję wszystkich znaków z wiersza z wyjątkiem ostatniego (pierwsza granica ma domyślną wartość 0). W obu przypadkach taki prosty wycinek spełnia zadanie, które w językach niższego rzędu musi być zaimplementowane w jawnym sposobie.

Warto przypomnieć, że bardzo często preferowanym rozwiązaniem jest usuwanie znaków nowego wiersza za pomocą metody `line.rstrip` — jej wywołanie pozostawia wiersz nietknięty, kiedy na jego końcu nie ma znaków nowego wiersza, co często zdarza się w przypadku plików tworzonych przez narzędzia do edycji tekstu. Zastosowanie wycinków ma sens tylko wtedy, kiedy wiadomo, że wiersz jest poprawnie zakończony.

Narzędzia do konwersji łańcuchów znaków

Jednym z fundamentalnych założeń projektowych Pythona jest to, że ten język ma zniechęcać użytkownika do zgadywania. Ewidentnym przykładem tej koncepcji jest niemożność dodawania do siebie liczb i łańcuchów znaków, nawet kiedy łańcuch wygląda jak liczba, to znaczy składa się z samych cyfr:

```
# Python 3.x
>>> "42" + 1
TypeError: Can't convert 'int' object to str implicitly
# Python 2.x
>>> "42" + 1
TypeError: cannot concatenate 'str' and 'int' objects
```

Jest to celowe działanie — ponieważ symbol `+` może oznaczać zarówno dodawanie, jak i konkatenację, wybór typu konwersji byłby niejednoznaczny, stąd wyrażenie takie traktowane jest jako błąd. Python z reguły stara się unikać niepotrzebnej magii, zwłaszcza jeżeli może nam ona skomplikować życie.

Co zatem można zrobić, gdy skrypt pobierze liczbę jako łańcuch tekstowy z pliku czy interfejsu użytkownika? Sztuczka polega na zastosowaniu odpowiednich narzędzi do konwersji przed potraktowaniem łańcucha jako liczby bądź odwrotnie.

```
>>> int("42"), str(42)                                # Konwersja z łańcucha i na
    łańcuch
(42, '42')
>>> repr(42)                                         # Konwersja na łańcuch w postaci
    kodu
'42'
```

Funkcja `int` przekształca łańcuch znaków na liczbę, natomiast funkcja `str` — liczbę na łańcuch znaków (czyli to, jak wygląda ona po wyświetleniu). Funkcja `repr` (i jej starszy odpowiednik — wyrażenie z apostrofem odwrotnym, usunięte w wersji 3.x) również dokonuje konwersji liczby na jej reprezentację łańcuchową, jednak zwraca obiekt w postaci łańcucha kodu, który można ponownie wykonać w celu odtworzenia obiektu. Wynik wyrażenia wyświetlony za pomocą funkcji `print` jest ujęty w apostrofy; sposób wywołania tej funkcji różni się nieco w zależności od wersji Pythona:

```

>>> print(str('mielonka'), repr('mielonka'))      # w wersji 2.x: print
str('mielonka'), repr('mielonka')

mielonka 'mielonka'

>>> str('mielonka'), repr('mielonka')           # Bezpośrednie wyświetlanie
wyniku

('mielonka', "'mielonka'")

```

Więcej informacji na ten temat można znaleźć w rozdziale 5., w ramce „Formaty wyświetlania – funkcje str() i repr()”. Zazwyczaj zalecanym sposobem konwersji liczb na ciągi znaków i odwrotnie jest zastosowanie funkcji int oraz str.

Choć nie można mieszać ze sobą łańcuchów znaków oraz liczb wokół operatorów takich, jak +, to jeżeli wystąpi taka potrzeba, można ręcznie przekonwertować odpowiednie operandy przed wykonaniem działania.

```

>>> S = "42"

>>> I = 1

>>> S + I

TypeError: Can't convert 'int' object to str implicitly

>>> int(S) + I                                # Wymuszenie dodawania

43

>>> S + str(I)                               # Wymuszenie konkatenacji

'421'

```

Podobne funkcje wbudowane obsługują konwersje liczb zmienoprzecinkowych na łańcuchy znaków i odwrotnie.

```

>>> str(3.1415), float("1.5")

('3.1415', 1.5)

>>> text = "1.234E-10"

>>> float(text)                            # W wersjach wcześniejszych niż 2.7 i 3.1
wyświetlanych było więcej cyfr

1.234e-10

```

Nieco później będziemy również omawiać wbudowaną funkcję eval, która interpretuje przekazany do niej łańcuch znaków jako ciąg zawierający kod wyrażenia w języku Python i wykonuje go, dzięki czemu może konwertować łańcuch znaków na obiekt dowolnego typu. Funkcje int i float dokonują konwersji jedynie na liczby, jednak ograniczenie to oznacza również, że są zazwyczaj szybsze (i bardziej bezpieczne, ponieważ nie przyjmują dowolnego kodu wyrażenia). Jak widzieliśmy w rozdziale 5., konwersji liczb na łańcuchy znaków można również dokonać za pomocą wyrażenia formatującego z łańcuchem znaków. Zostanie to omówione w dalszej części rozdziału.

Konwersje kodu znaków

Jeżeli chodzi o konwersje, możemy również przekształcić pojedynczy znak na odpowiadający mu kod liczbowy (np. w zestawie ASCII), przekazując go do wbudowanej funkcji ord. Zwraca ona wartość kodu liczbowego odpowiadającego danemu znakowi w pamięci. Funkcja chr wykonuje operację odwrotną, przyjmując kod liczbowy ASCII i przekształcając go na odpowiadający mu znak.

```
>>> ord('s')
115
>>> chr(115)
's'
```

Technicznie rzecz biorąc, obie funkcje konwertują znaki na i z ich kodów Unicode, które są tylko ich identyfikatorami w podstawowym zestawie znaków. W przypadku tekstu ASCII jest to dobrze nam już znana 7-bitowa liczba całkowita, która mieści się w jednym bajcie w pamięci, ale zakres kodów znaków dla innych rodzajów tekstu Unicode może być znacznie szerszy (więcej na temat zestawów znaków i kodowania Unicode znajdziesz w rozdziale 37.). Aby zastosować takie funkcje do wszystkich znaków w ciągu, możesz użyć pętli. Narzędzi tych można również użyć do wykonania pewnego rodzaju matematyki opartej na łańcuchach. Na przykład, aby przejść do następnego znaku, przekonwertuj i wykonaj następujące obliczenia matematyczne na liczbach całkowitych:

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

W przypadku ciągów jednoznakowych stanowi to alternatywę dla korzystania z wbudowanej funkcji `int` do konwersji ciągów na liczby całkowite (choć ma to sens tylko w zestawach znaków, w których elementy są uporządkowane zgodnie z oczekiwaniemi Twojego programu!):

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```

Taką konwersję można wykorzystać w połączeniu z pętlami (wprowadzonymi w rozdziale 4. i omówionymi szczegółowo w następnej części tej książki) do przekształcenia łańcucha cyfr w zapisie dwójkowym na odpowiadające im wartości liczb w zapisie dziesiętnym. Przy każdej iteracji pętli aktualna wartość mnożona jest przez 2 i dodawana do wartości liczbowej kolejnej cyfry.

```
>>> B = '1101'      # Przekształcenie łańcucha znaków w notacji dwójkowej na
                  # liczbę całkowitą z użyciem funkcji ord()
>>> I = 0
>>> while B: != '':
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
>>> I
```

Podobny efekt do mnożenia przez 2 miałaby operacja przesunięcia bitowego w lewo ($I \ll 1$). Taką modyfikację pozostawimy jako sugerowane ćwiczenie do samodzielnego wykonania, ponieważ jeszcze szczegółowo nie omawialiśmy pętli, ale też dlatego, że w Pythonie 2.6, 3.0 i nowszych przekształcenia z notacji dwójkowej na liczbę całkowitą i z powrotem można dokonać z użyciem funkcji `int` i `bin` omówionych w rozdziale 5.

```
>>> int('1101', 2)          # Przekształcenie liczby binarnej na liczbę
całkowitą: funkcja wbudowana
```

13

```
>>> bin(13)                # Przekształcenie liczby całkowitej na liczbę
binarną: funkcja wbudowana
'0b1101'
```

Jak widać, deweloperzy Pythona mają skłonność do automatyzacji powszechnie wykonywanych zadań, wystarczy tylko dać im trochę czasu!

Modyfikowanie łańcuchów znaków

Warto przypomnieć tutaj pojęcie „sekwencji niemutowalnej”. Określenie *niemutowalna* oznacza, że nie można zmienić łańcucha znaków w miejscu (na przykład poprzez przypisanie nowej wartości do elementu o wybranym indeksie):

```
>>> S = 'mielonka'
>>> S[0] = "x"           # Zgłasza błąd!
TypeError: 'str' object does not support item assignment
```

W jaki sposób można zatem zmodyfikować w Pythonie informacje tekstowe? Aby zmienić łańcuch znaków, generalnie powinieneś za pomocą odpowiednich narzędzi, takich jak konkatenacja czy wycinki, utworzyć nowy łańcuch znaków, a następnie przypisać rezultat z powrotem do oryginalnej zmiennej (nazwy).

```
>>> S = S + 'MIELONKA!'          # Aby zmienić łańcuch, należy
utworzyć nowy
>>> S
'mielonkaMIELONKA!'
>>> S = S[:8] + 'Jajka' + S[-1]
>>> S
'mielonkaJajka!'
```

W pierwszym przykładzie na końcu łańcucha `S` dodany zostaje (dzięki konkatenacji) nowy podciąg znaków (tak naprawdę działanie to tworzy nowy łańcuch znaków i przypisuje go z powrotem do zmiennej `S`, ale możemy to sobie wyobrazić jako „modyfikację” oryginalnego łańcucha). W drugim przykładzie osiem znaków zostaje zastąpionych pięcioma za pomocą wycinka, indeksowania i konkatenacji. Jak zobaczymy w dalszej części rozdziału, podobny rezultat można osiągnąć za pomocą wywołania metod, takich jak `replace`:

```
>>> S = 'mielonka'
>>> S = S.replace('lon', 'lonecz')
>>> S
```

'mieloneczka'

Tak jak wszystkie operacje zwracające nowe wartości łańcuchów znaków, metody łańcuchów znaków generują nowe obiekty. Jeżeli chcemy je zachować, można je przypisać do zmiennej. Wygenerowanie nowego obiektu łańcucha znaków dla każdej modyfikacji tego łańcucha nie jest aż tak niewydajne, jak może się na pierwszy rzut oka wydawać — powinieneś pamiętać, że — jak wspomnieliśmy w poprzednim rozdziale — Python automatycznie zwalnia stare, nieużywane obiekty łańcuchów znaków w miarę postępów programu, dzięki czemu nowsze obiekty mogą wykorzystywać przestrzeń zajmowaną wcześniej przez inne obiekty. Python zazwyczaj jest znacznie bardziej efektywny, niż moglibyśmy się tego spodziewać.

Wreszcie, możemy również budować nowe wartości tekstowe za pomocą wyrażeń formatujących z łańcuchami znaków. Obydwa poniższe przykłady dokonują podstawienia obiektów do łańcucha znaków, to znaczy przekształcają obiekty na łańcuchy znaków i zmieniają oryginalny łańcuch znaków zgodnie ze specyfikacją formatowania.

```
>>> 'Ten %d %s jest martwy!' % (1, 'ptak')      # Wyrażenie formatujące:  
wszystkie wersje Pythona  
  
'Ten 1 ptak jest martwy!'  
  
>>> 'Ten {0} {1} jest martwy!'.format(1, 'ptak')  # Metoda formatująca w  
wersjach 2.6, 2.7 i 3.x  
  
'Ten 1 ptak jest martwy!'
```

Warto tutaj zauważyć, że pomimo tego, co napisaliśmy nieco wcześniej o podstawianiu, w obu przypadkach wynikiem formatowania są nowe, a nie zmodyfikowane obiekty łańcuchowe. Formatowanie omówimy bardziej szczegółowo w dalszej części rozdziału; jak się sam przekonasz, to narzędzie jest znacznie bardziej użyteczne, niż sugerowałby powyższy przykład. Ponieważ drugi z powyższych przykładów korzysta z wywołania metody `format`, to zanim przejdziemy do dalszego formatowania, zapoznamy się nieco bliżej z wywołaniami metod łańcuchów znaków.



Jak już wspominaliśmy w rozdziale 4. (i opowiem o tym jeszcze więcej w rozdziale 37.), w Pythonie 2.6 i 3.0 wprowadzono nowy typ łańcuchów znaków, znany jako `bytearray`, który jest mutowalny i może być zmieniany w miejscu. Obiekty typu `bytearray` naprawdę nie są łańcuchami znaków; zamiast tego są sekwencjami 8-bitowych liczb całkowitych. Nie zmienia to jednak w niczym faktu, że obsługują większość tych samych operacji co zwykłe łańcuchy znaków i można je wyświetlać w taki sam sposób jak łańcuchy znaków ASCII. Z tego względu zastosowanie takich obiektów jest kolejnym sposobem pozwalającym na przetwarzanie dużych ilości prostego, 8-bitowego tekstu, który musi być często modyfikowany (bardziej rozbudowane schematy kodowania tekstu, takie jak Unicode, wymagają zastosowania innych rozwiązań). W rozdziale 37. pokażemy również, że funkcje `ord` i `chr` obsługują także znaki Unicode, których często nie można zapisywać w pojedynczych bajtach.

Metody łańcuchów znaków

Poza operatorami wyrażeń łańcuchy znaków udostępniają zbiór *metod* implementujących bardziej wyszukane zadania związane z przetwarzaniem tekstu. W Pythonie wyrażenia i funkcje wbudowane mogą działać na różnych typach obiektów, ale metody są *specyficzne dla typów obiektów* — na przykład metody łańcuchów znaków działają tylko na obiektach łańcuchów znaków. Zestawy metod niektórych typów w Pythonie 3.x nakładają się na siebie (na przykład

wiele typów posiada metody `count` i `copy`), ale mimo to metody są bardziej specyficzne dla typów niż większość innych narzędzi.

Składnia wywoływania metod

Jak wspominaliśmy w rozdziale 4., metody to po prostu funkcje powiązane z określonymi obiektami i wykonujące na nich różne operacje. Technicznie są to atrybuty przypisane do obiektów, które mają odniesienie do wywoływanych funkcji. Mówiąc bardziej szczegółowo, funkcje to pakiety kodu, a wywołania metod łączą w sobie dwie operacje jednocześnie — pobieranie atrybutu i wywołanie funkcji:

Pobieranie atrybutu

Wyrażenie w postaci `obiekt.atrybut` oznacza: „pobierz wartość `atrybutu` z obiektu”.

Wywołanie

Wyrażenie w postaci `funkcja(argumenty)` oznacza: „wywołaj kod `funkcji`, przekazując jej zero lub większą liczbę rozdzielonych przecinkami obiektów `argumentów` i zwróć wyniki działania `funkcji`”.

Połączenie ze sobą tych dwóch elementów pozwala na wywołanie metody obiektu. Metody są wywoływanie za pomocą następującego wyrażenia:

`obiekt.metoda(argumenty)`

które jest przetwarzane od lewej strony do prawej. Oznacza to, że Python najpierw pobiera metodę obiektu, a następnie wywołuje ją, przekazując do niej zarówno sam `obiekt`, jak i `argumenty`. Inaczej mówiąc, wyrażenie wywołujące metodę oznacza mniej więcej tyle:

Wywołaj metodę, która będzie przetwarzała obiekt razem z argumentami.

Jeżeli metoda wykonuje jakieś obliczenia, ich wynik zostanie zwrócony jako wynik całego wyrażenia wywołującego metodę. Na przykład:

```
>>> S = 'spam'  
>>> result = S.find('pa')      # Wywołanie metody find w celu wyszukania  
ciągu znaków 'pa' w łańcuchu S
```

Takie podejście sprawdza się zarówno w przypadku metod wbudowanych typów, jak i klas zdefiniowanych przez użytkownika, o których będziemy mówić później. Jak zobaczymy w tej części książki, większość obiektów posiada swoje wywoływalne metody i w każdym przypadku ogólna składnia wywołania jest taka sama. Jak się okazuje, aby wywołać metodę obiektu, trzeba się odwołać do jego nazwy; metody nie mogą być uruchamiane (i nie ma to większego sensu) bez odwołania do określonego obiektu.

Metody typów znakowych

W tabeli 7.3 podsumowano metody i wzorce wywoływania większości metod obiektów wbudowanych typów znakowych w Pythonie 3.x. Metody takich typów często ulegają zmianom, warto więc zerknąć do dokumentacji Pythona w celu przejrzenia najbardziej aktualnej listy metod lub w sesji interaktywnej skorzystać z funkcji `dir` lub `help` dla wybranego łańcucha znaków lub ogólnie dla typu `str`. W Pythonie 2.x metody łańcuchów znaków trochę się różnią, na przykład `decode`, w której obsługa znaków Unicode jest nieco inna (co omówimy w rozdziale 37.). W tej tabeli `S` reprezentuje obiekt typu `string`, opcjonalne argumenty zostały ujęte w nawiasy kwadratowe. Metody łańcuchów znaków znajdujące się w tabeli implementują operacje wyższego poziomu, takie jak dzielenie i łączenie, zmiana wielkości liter, sprawdzanie zawartości oraz wyszukiwanie i zastępowanie podłańcuchów znaków.

Tabela 7.3. Wywołania metod łańcuchów znaków w Pythonie 3.0

S.capitalize()	S.ljust(width [, fill])
S.casefold()	S.lower()
S.center(width [, fill])	S.lstrip([chars])
S.count(sub [, start [, end]])	S.maketrans(x[, y[, z]])
S.encode([encoding [,errors]])	S.partition(sep)
S.endswith(suffix [, start [, end]])	S.replace(old, new [, count])
S.expandtabs([tabsize])	S.rfind(sub [,start [,end]])
S.find(sub [, start [, end]])	S.rindex(sub [, start [, end]])
S.format(fmtstr, *args, **kwargs)	S.rjust(width [, fill])
S.index(sub [, start [, end]])	S.rpartition(sep)
S.isalnum()	S.rsplit([sep[, maxsplit]])
S.isalpha()	S.rstrip([chars])
S.isdecimal()	S.split([sep [,maxsplit]])
S.isdigit()	S.splitlines([keepends])
S.isidentifier()	S.startswith(prefix [, start [, end]])
S.islower()	S.strip([chars])
S.isnumeric()	S.swapcase()
S.isprintable()	S.title()
S.isspace()	S.translate(map)
S.istitle()	S.upper()
S.isupper()	S.zfill(width)
S.join(iterable)	

Jak widać, istnieje spora liczba metod znakowych i nie mamy tu miejsca, aby je wszystkie omówić; więcej szczegółowych informacji na ich temat znajdziesz w dokumentacji Pythona. Aby jednak pomóc Ci w rozpoczęciu pracy, przyjrzymy się niebawem kilku przykładom kodu, które pokazują sposoby użycia niektórych częściej stosowanych metod i przy okazji ilustrują podstawy przetwarzania tekstu w Pythonie.

Przykłady metod łańcuchów znaków – modyfikowanie

Jak już kilka razy wspominaliśmy, łańcuchy znaków są niemutowalne, zatem nie można ich bezpośrednio zmieniać w miejscu. W wersjach 2.6, 3.0 i nowszych istnieje typ `bytearray`, który obsługuje takie operacje, ale tylko dla prostych typów 8-bitowych. Co prawda zagadnienia związane z modyfikowaniem łańcuchów tekstów omawialiśmy już wcześniej, ale mimo to wróćmy do tego na chwilę w kontekście metod ciągów.

Ogólnie rzecz biorąc, aby z istniejącego łańcucha utworzyć nową wartość tekstową, musimy utworzyć nowy łańcuch za pomocą operacji takich jak wycinki czy konkatenacja. Na przykład, żeby zastąpić znaki w środku łańcucha, możesz użyć kodu pokazanego poniżej:

```
>>> S = 'jabłka'  
>>> S = S[:2] + 'bł' + S[3:]      # Wycinamy wybrane sekcje z S  
>>> S  
'jabłka'
```

Jeżeli jednak naprawdę chcesz tylko zastąpić podciąg znaków, możesz skorzystać z metody `replace`.

```
>>> S = 'jabłka'  
>>> S = S.replace('jk', 'błk')    # Zamieniamy wszystkie wystąpienia 'jk' na  
'błk' w ciągu S  
>>> S  
'jabłka'
```

Metoda `replace` jest bardziej ogólna, niż na to wskazuje powyższy przykład. Jako argumenty przyjmuje poszukiwany podciąg znaków (dowolnej długości) oraz ciąg docelowy (również dowolnej długości), którym zostanie on zastąpiony, a następnie wykonuje globalne wyszukiwanie i zastępowanie ciągów znaków.

```
>>> 'aa$bb$cc$dd'.replace('$', 'JAJKA')  
'aaJAJKAbbJAJKAccJAJKAdd'
```

W takiej roli metodę `replace` można wykorzystać jako narzędzie implementujące zastąpienia szablonów (na przykład listów z formularzami). Warto zauważyć, że tym razem po prostu wyświetlamy wynik, zamiast przypisywać go do zmiennej — przypisanie do zmiennej jest konieczne tylko wtedy, gdy chcesz zachować wynik do dalszej pracy.

Jeżeli chcesz zastąpić jeden podciąg znaków o określonym rozmiarze, który może się pojawić w dowolnym miejscu łańcucha, możesz albo ponownie skorzystać z metody `replace`, albo najpierw odszukać podłańcuch za pomocą metody `find`, a później skorzystać z wycinka.

```
>>> S = 'xxxxJAJKAxXXXXJAJKAxXXXX'  
>>> where = S.find('JAJKA')                      # Odszukanie pozycji  
>>> where                                         # Występuje z przesunięciem 4  
4  
>>> S = S[:where] + 'MIELONKA' + S[(where+4):]  
>>> S  
'xxxxMIELONKAXXXXXJAJKAxXXXX'
```

Metoda `find` zwraca wartość pozycji przesunięcia, na której znajduje się podciąg (domyślnie wyszukiwanie odbywa się od początku łańcucha), lub `-1`, jeśli nie zostanie on odnaleziony. Jak

zauważliśmy wcześniej, jest to operacja wyszukiwania podłańcuchów znaków działająca podobnie do wyrażenia `in`, ale zwracająca pozycję znalezionej podłańcuchu.

Inną opcją jest skorzystanie z metody `replace` w połączeniu z trzecim argumentem, który ograniczy ją do jednego zastąpienia.

```
>>> S = 'xxxxJAJKAxxxxxJAJKAxxxxx'  
>>> S.replace('JAJKa', 'MIELONKA') # Zastępuje wszystkie  
wystąpienia  
'xxxxMIELONKAxxxxxMIELONKAxxxxx'  
>>> S.replace('JAJKa', 'MIELONKA', 1) # Zastępuje jedno wystąpienie  
'xxxxMIELONKAxxxxxJAJKAxxxxx'
```

Warto zwrócić uwagę na to, że metoda `replace` zwraca za każdym razem nowy obiekt łańcucha znaków. Ponieważ łańcuchy znaków są niemutowalne, metody nigdy tak naprawdę nie modyfikują ich w miejscu, nawet jeżeli ich nazwy (z ang. `replace` — zastąp) mogą to sugerować.

To, że operacje konkatenacji i metoda `replace` generują za każdym razem nowe obiekty łańcuchów znaków, może być potencjalną wadą używania ich do zastępowania łańcuchów. Jeżeli w jednym łańcuchu znaków musisz wprowadzić wiele zmian, być może uda Ci się poprawić wydajność skryptu, przekształcając łańcuch znaków na obiekt mutowalny, obsługujący zmiany w miejscu.

```
>>> S = 'brama'  
>>> L = list(S)  
>>> L  
['b', 'r', 'a', 'm', 'a']
```

Wbudowana funkcja `list` (wywołanie konstruktora obiektu) tworzy nową listę z elementów dowolnej sekwencji — w tym przypadku rozbijając znaki łańcucha na listę. Kiedy łańcuch będzie miał taką postać, możesz wprowadzić do niego wiele zmian bez generowania za każdym razem nowej jego kopii.

```
>>> L[2] = 'e' # Działa dla list, ale nie dla  
łańcuchów znaków  
>>> L[3] = 'j'  
>>> L  
['b', 'r', 'e', 'j', 'a']
```

Jeżeli po tych zmianach musisz przekształcić obiekt z powrotem na łańcuch znaków (na przykład w celu zapisania go do pliku), powinieneś skorzystać z metody `join`, która złoży listę z powrotem w łańcuch.

```
>>> S = ''.join(L)  
>>> S  
'breja'
```

Metoda `join` może na pierwszy rzut oka wyglądać nieco dziwnie. Ponieważ jest metodą łańcuchów znaków (a nie list), wywoływana jest z użyciem żądanego separatora. Metoda ta łączy ze sobą łańcuchy podane w postaci listy (lub innego obiektu iterowanego), wstawiając pomiędzy nie podany separator. W tym przypadku wykorzystuje jako separator pusty łańcuch znaków, pozwalający na przekształcenie listy z powrotem w łańcuch znaków. Metoda ta ma

jednak bardziej uniwersalne zastosowanie i może działać z dowolnym separatorem łańcucha i dowolnym, iterowalnym obiektem znakowym.

```
>>> 'JAJKA'.join(['mielonka', 'ser', 'szynka', 'tost'])  
'mielonkaJAJKAserJAJKAszynkaJAJKAtost'
```

W rzeczywistości łączenie ciągów znaków w ten sposób zazwyczaj będzie znacznie szybsze od konkatenacji indywidualnych łańcuchów. Warto również zwrócić uwagę na wspomniany wyżej typ `bytearray` wprowadzony w Pythonie 3.0 i 2.6 (omówiony szerzej w rozdziale 37.) — ponieważ może on być modyfikowany w miejscu, stanowi swego rodzaju alternatywę dla metody `join`, co może być przydatne w przypadku niektórych rodzajów 8-bitowego tekstu, który musi być często modyfikowany.

Przykłady metod łańcuchów znaków — analiza składniowa tekstu

Kolejnym, często spotykanym zadaniem metod łańcuchów znaków jest *analiza składniowa* tekstu, czyli analiza jego struktury i wyodrębnianie wybranych podłańcuchów. Aby pobrać podłańcuch znajdujący się na określonych pozycjach przesunięcia, można skorzystać z wycinków.

```
>>> line = 'aaa bbb ccc'  
>>> col1 = line[0:3]  
>>> col3 = line[8:]  
>>> col1  
'aaa'  
>>> col3  
'ccc'
```

W powyższym przykładzie kolumny danych mają stałe wartości przesunięcia, dzięki czemu można je łatwo wyciąć z oryginalnego łańcucha znaków. Taka technika nadaje się do analizy składniowej tekstu dopóty, dopóki komponenty naszych danych znajdują się na stałych pozycjach. Gdyby zamiast tego dane rozdzielały jakiś rodzaj separatora, można komponenty łańcucha pobrać po rozdzieleniu na elementy składowe. Będzie to działać nawet wtedy, gdy dane występują w łańcuchu na zmiennych pozycjach.

```
>>> line = 'aaa bbb ccc'  
>>> cols = line.split()  
>>> cols  
['aaa', 'bbb', 'ccc']
```

Metoda łańcuchów znaków `split` dzieli łańcuch na listę podłańcuchów według separatora. W poprzednim przykładzie nie przekazaliśmy ogranicznika, przez co użyta została wartość domyślna, która jest dowolny biały znak. Łańcuch znaków jest dzielony w miejscu wystąpienia jednej bądź kilku spacji, tabulatorów lub nowych wierszy. Metoda ta zwraca listę wynikowych podłańcuchów. Można ją również zastosować w połączeniu z innymi separatorami. Poniższy przykład dzieli (i tym samym analizuje składniowo) łańcuch znaków w miejscu wystąpienia przecinka, który często rozdziela dane zwarcane przez wiele narzędzi baz danych.

```
>>> line = 'amadeusz,haker,40'
```

```
>>> line.split(',')
['amadeusz', 'haker', '40']
```

Separatory mogą składać się z większej liczby znaków, tak jak to zostało pokazane poniżej.

```
>>> line = "jestemMIELONKAdrwalemMIELONKAiMIELONKAjestemMIELONKAOK"
>>> line.split("MIELONKA")
['jestem', 'drwalem', 'i', 'jestem', 'OK']
```

Choć istnieją ograniczenia co do potencjału wycinków i podziałów w analizie składniowej, obie operacje działają bardzo szybko i świetnie sobie radzą z podstawowymi zadaniami wymagającymi ekstrakcji tekstu. Dane tekstowe rozdzielane przecinkami są wykorzystywane przy zapisywaniu plików w formacie CSV; bardziej zaawansowane narzędzia tego typu można znaleźć w module `csv` w standardowej bibliotece Pythona.

Inne często używane metody łańcuchów znaków

Pozostałe metody łańcuchów znaków mają bardziej wyspecjalizowane role. Służą na przykład do odcinania białych znaków na końcu wiersza, zmiany wielkości liter, sprawdzania zawartości łańcuchów znaków czy istnienia danego podłańcucha na końcu.

```
>>> line = "Rycerze, którzy mówią Ni!\n"
>>> line.rstrip()
'Rycerze, którzy mówią Ni!'
>>> line.upper()
'RYCERZE, KTÓRZY MÓWIĄ NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\\n')
True
>>> line.startswith('Rycerze')
True
```

Czasami do uzyskania tych samych efektów co za pomocą metod łańcuchów znaków można wykorzystać również alternatywne techniki. Na przykład, aby sprawdzić obecność podłańcucha znaków, można wykorzystać operator `in`. Połączenie operacji obliczenia długości i sporządzenia wycinków może natomiast zastąpić metodę `endswith`.

```
>>> line
'Rycerze, którzy mówią Ni!\n'
>>> line.find('Ni') != -1          # Szukanie za pomocą wywołania metody lub
                                 # wyrażenia
True
>>> 'Ni' in line
True
```

```

>>> sub = 'Ni!\n'
>>> line.endswith(sub)           # Sprawdzenie końca łańcucha za pomocą
wywołania metody lub wycinka
True
>>> line[-len(sub):] == sub
True

```

Zachęcam również do zapoznania się z metodami formatowania opisymi w dalszej części rozdziału. Znajdziesz tam opisy narzędzi przeznaczonych do podstawiania łańcuchów znaków, które łączą wiele operacji w jednym wywołaniu.

Ponieważ istnieje tak wiele metod łańcuchów znaków, nie będziemy ich tutaj omawiać po kolei. Kilka dodatkowych przykładów z łańcuchami znaków znajdziesz w dalszej części książki, a więcej szczegółowych informacji znajdziesz w dokumentacji biblioteki Pythona czy innych źródłach; powinieneś również samodzielnie eksperymentować w sesji interaktywnej. Aby skorzystać z wbudowanej pomocy Pythona, możesz użyć funkcji `help(S.metoda)`.

Warto podkreślić, że żadna z metod łańcuchów znaków nie przyjmuje wzorców. Do przetwarzania tekstu opartego na wzorcach należy użyć modułu wyrażeń regularnych `re` z biblioteki standardowej Pythona — zaawansowanego narzędzia przedstawionego w rozdziale 4., którego opis wykracza jednak poza zakres naszej książki (dodatkowy przykład znajdziesz pod koniec rozdziału 37.). Ze względu na brak obsługi wzorców metody łańcuchów znaków z reguły działają nieco szybciej niż narzędzia z modułu `re`.

Oryginalny moduł string (usunięty w wersji 3.0)

Historia metod łańcuchów znaków Pythona jest nieco zagmatwana. Mniej więcej przez pierwszą dekadę istnienia języka Python udostępniany był moduł biblioteki standardowej o nazwie `string`. Zawierał on funkcje, które w dużej mierze odzwierciedlały obecny zbiór metod obiektów łańcuchów znaków. W odpowiedzi na liczne prośby użytkowników, w Pythonie 2.0 funkcje te zostały udostępnione jako metody obiektów łańcuchów znaków. Ponieważ jednak do tej pory bardzo wielu programistów napisało już duże ilości kodu w oparciu o oryginalny moduł `string`, pozostał on w bibliotece w celu zachowania kompatybilności wstępnej.

Obecnie powinniśmy korzystać *jedynie z metod łańcuchów znaków*, a nie z oryginalnego modułu `string`. Tak naprawdę wywołania z oryginalnego modułu odpowiadające dzisiejszym metodom łańcuchów znaków zostały zupełnie usunięte w wersjach 3.x Pythona i nie należy ich używać w nowym kodzie w wersjach 2.x i 3.x. Ponieważ jednak nadal można napotkać kod korzystający z tego modułu w starszych programach dla wersji 2.x, a nasza książka obejmuje zarówno wersje 2.x, jak i 3.x, warto zamieścić tu kilka słów na jego temat.

Rezultatem tych pozostałości jest to, że w Pythonie 2.x nadal istnieją dwa sposoby wywoływania bardziej zaawansowanych operacji na łańcuchach znaków — poprzez wywołanie metod obiektu oraz poprzez wywołanie funkcji modułu `string` i przekazanie obiektu jako argumentu. Mając na przykład zmienną `X` przypisaną do obiektu łańcucha znaków, wywołanie metody obiektu:

```
X.metoda(argumenty)
```

jest zazwyczaj odpowiednikiem wywołania tej samej operacji za pośrednictwem modułu `string` (o ile oczywiście najpierw ten moduł zimportowaliśmy):

```
string.metoda(X, argumenty)
```

Poniżej znajduje się przykład zastosowania metody łańcucha znaków.

```
>>> S = 'a+b+c+'
```

```
>>> x = S.replace('+', 'jajka')
>>> x
'ajajkabjajkacjajka'
```

Aby uzyskać dostęp do tej samej operacji za pośrednictwem modułu `string` w Pythonie 2.x, należy zimportować ten moduł (przynajmniej raz na sesję) i przekazać mu obiekt.

```
>>> import string
>>> y = string.replace(S, '+', 'jajka')
>>> y
'ajajkabjajkacjajka'
```

Ponieważ korzystanie z modułu `string` było standardem przez wiele lat, a także dlatego, że łańcuchy znaków są tak ważnym komponentem większości programów, w kodzie napisanym w Pythonie 2.x można spotkać oba wzorce wywołania.

Ponownie trzeba jednak podkreślić, że obecnie należy korzystać z wywoływanego metod obiektów, a nie z modułu `string`. Ma to swoje uzasadnienie związane nie tylko z tym, że obsługa wywołania tego modułu zniknęła w Pythonie 3.x. Po pierwsze, wywoływanie modułu wymaga wcześniejszego zimportowania go (w przypadku metod nie jest to konieczne). Po drugie, moduł sprawia, że całe wywołanie robi się o kilka znaków dłuższe (jest tak wtedy, gdy moduł ładowamy za pomocą instrukcji `import`, a nie `from`). I wreszcie — moduł działa wolniej od metod, ponieważ odwzorowuje wywołania z powrotem na metody, co wprowadza niepotrzebne dodatkowe wywołanie.

Oryginalny moduł `string`, pozbawiony co prawda ekwiwalentów metod łańcuchowych, został zachowany w Pythonie 3.x, ponieważ zawiera dodatkowe narzędzia, w tym predefiniowane stałe ciągów (np. `string.digits`) i system obiektów `Template` — dosyć tajemnicze narzędzie formatujące, poprzedzające powstanie metody `format`, którego nie będziemy omawiać (szczegółowe informacje można znaleźć w krótkiej notatce porównującej takie obiekty z innymi narzędziami do formatowania oraz w dokumentacji biblioteki Pythona). Jeżeli naprawdę chcesz zmienić swój kod Pythona 2.x tak, aby używać go w wersji 3.x, powinieneś uznać wywołania funkcji z modułu `string` za duchy przeszłości.

Wyrażenia formatujące łańcuchy znaków

Omówione wyżej metody łańcuchów znaków i operacje sekwencji dają duże możliwości, ale Python oferuje bardziej zaawansowane sposoby realizacji zadań związanych z łańcuchami znaków: *formatowanie łańcuchów znaków* pozwala nam przeprowadzić wiele podstawień w łańcuchu znaków w pojedynczej operacji. Użycie tych narzędzi nigdy nie jest niezbędne do realizacji zadania, ale często stanowi wygodne rozwiązanie, szczególnie przy formatowaniu łańcuchów znaków wyświetlanych użytkownikowi programu. W efekcie ogromnej ilości nowych pomysłów pojawiających się w świecie Pythona formatowanie ciągów jest dziś dostępne w Pythonie w dwóch odmianach (nie licząc rzadziej używanego systemu szablonów `Template` dostępnego w module `string`, o którym wspominaliśmy w poprzedniej sekcji):

Formatowanie łańcuchów znaków — wyrażenia formatujące: '...%s...' % (wartości)

Oryginalna technika formatowania łańcuchów znaków, dostępna w Pythonie od zawsze; oparta jest na modelu użytym w funkcji `sprintf` języka C. Przykłady użycia tej metody można znaleźć praktycznie w każdym programie napisanym w Pythonie.

Formatowanie łańcuchów znaków — wywołanie metody: '...{}...'.format(wartości)

Nowsza technika dodana w Pythonie 2.6 i 3.0. To rozwiązanie pochodzi częściowo z narzędzi o tej samej nazwie w C# / .NET i jego możliwości w znacznym stopniu pokrywają się z funkcjonalnością wyrażeń formatujących.

Choć formatowanie z wykorzystaniem wywołania metody `format` jest rozwiązaniem znacznie nowszym, istnieje pewne prawdopodobieństwo, że z czasem jedno lub drugie rozwiązanie zostanie uznane za przestarzałe i wcześniej czy później zostanie usunięte. Kiedy w roku 2008 pojawiła się wersja 3.0, wydawało się już, że mechanizm wyrażeń formatujących jest skazany na przegrana i zniknie z późniejszych wersji Pythona. Rzeczywiście, dokumentacja wersji 3.0 informowała o planach wycofania tego mechanizmu w wersji 3.1 i jego późniejszym usunięciu. Tak się jednak nie stało ani do roku 2013, kiedy pojawiła się wersja 3.3, ani później, a dzisiaj wydaje się to równie mało prawdopodobne, biorąc pod uwagę szerokie zastosowanie wyrażeń formatujących — wystarczy wspomnieć, że są one nadal powszechnie stosowane w standardowej bibliotece Pythona!

Oczywiście dalsze losy tych mechanizmów w dużej mierze zależą od ich popularności wśród użytkowników Pythona. Z drugiej strony, ponieważ oba sposoby formatowania są poprawne i każdy z nich może pojawić się w kodzie, z którym się spotkasz, w tej książce w pełni omówimy obie techniki. Jak się przekonasz, oba rozwiązania są w dużej mierze *oparte na podobnych założeniach*, chociaż metoda `format` ma kilka dodatkowych funkcji (takich jak separatory tysięcy), a z kolei składnia wyrażeń formatujących jest zazwyczaj bardziej zwięzła i wydaje się być naturalnym rozwiązaniem dla większości programistów Pythona.

W przykładach omawianych w książce zaprezentujemy obie techniki w celach ilustracyjnych. Nawet jeśli mam jakieś swoje preferencje w tej materii, to będąc autorem tej książki, postaram się zachować to w tajemnicy i pozwolę sobie tylko zacytować jedną z sentencji zen Pythona, które możesz wyświetlić na ekranie za pomocą polecenia `import this`:

Powinien być jeden – i najlepiej tylko jeden – oczywisty sposób na zrobienie danej rzeczy.

Skoro jednak nowsza z metod formatowania ciągów nie jest zdecydowanie lepsza niż oryginalne i powszechnie używane wyrażenia formatujące, obarczanie programistów Pythona koniecznością poznania nowego, rozbudowanego rozwiązania w tej dziedzinie wydaje się być nieuzasadnione, a nawet — używając pierwotnego i ogólnie przyjętego znaczenia tego żargonowego określenia — „niepythonowe”. Programiści nie powinni być zmuszani do uczenia się dwóch skomplikowanych narzędzi, jeżeli ich funkcjonalność w dużym stopniu się pokrywa. W takiej sytuacji będziesz musiał sam ocenić, czy nowe rozwiązanie formatowania zasługuje na poświęcenie mu dodatkowej uwagi, a żeby Ci to ułatwić, w dalszej części tego rozdziału szczegółowo omówimy oba sposoby.

Formatowanie łańcuchów tekstu z użyciem wyrażeń formatujących — podstawa

Ponieważ *wyrażenia formatujące* ciągi znaków pojawiły się w Pythonie jako pierwsze, zaczniemy właśnie od nich. W Pythonie dostępny jest operator binarny `%`, który działa na łańcuchach znaków (jak pamiętamy, jest to również operator dzielenia z resztą, czy inaczej mówiąc, dzielenia modulo dla liczb). Po zastosowaniu do łańcuchów znaków operator `%` udostępnia łatwy sposób formatowania łańcuchów znaków, zgodnie z podanym wzorcem. Krótko mówiąc, operator `%` pozwala na zastępowanie wielu wystąpień podciągów znaków w jednym wywołaniu bez konieczności mozolnego składania ciągu wynikowego z poszczególnych elementów.

Aby formatować łańcuchy znaków:

1. Po lewej stronie operatora `%` umieszcza się łańcuch znaków, który będzie formatowany, zawierający jeden lub większą liczbę osadzonych znaczników

konwersji, z których każdy rozpoczyna się od znaku % (na przykład %d).

2. Po prawej stronie operatora % umieszczamy obiekt (lub obiekty, zapisane w krotce), który Python ma wstawić do formatowanego łańcucha w miejsce znaczników konwersji.

W przykładzie, który pokazaliśmy we wcześniejszej części rozdziału, znacznik %d w formatowanym łańcuchu znaków zostaje zastąpiony liczbą 1, natomiast znacznik %s będzie zastąpiony ciągiem znaków 'ptak'. W rezultacie otrzymujemy nowy łańcuch znaków zawierający te dwa podstawienia, który możemy wyświetlić lub zapisać do wykorzystania w innych celach.

```
>>> 'Ten %d %s jest martwy!' % (1, 'ptak') # Wyrażenie formatujące  
Ten 1 ptak jest martwy!
```

Z technicznego punktu widzenia wyrażenia formatujące łańcuchy znaków są zazwyczaj opcjonalne — to samo można najczęściej uzyskać za pomocą kilku konkatenacji i konwersji. Formatowanie pozwala jednak połączyć kilka kroków w jedną operację. Wyrażenia formatujące mają na tyle duże możliwości, że zasługują co najmniej na kilka kolejnych przykładów.

```
>>> exclamation = "Ni"  
>>> "Rycerze, którzy mówią %s!" % exclamation # Podstawianie łańcucha  
znaków  
'Rycerze, którzy mówią Ni!'  
>>> "%d %s %d you" % (1, 'spam', 4) # Podstawienia zależne od  
typu  
'1 spam 4 you'  
>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3]) # Wszystkie typy pasują do  
znacznika %s  
'42 -- 3.14159 -- [1, 2, 3]'
```

Pierwszy z powyższych przykładów wstawia łańcuch znaków "Ni" w miejsce po lewej stronie, zastępując znacznik %s. W drugim przykładzie do docelowego łańcucha znaków wstawiane są cztery wartości. Warto zauważyć, że kiedy do łańcucha wstawia się większą liczbę wartości, konieczne jest zgrupowanie ich w nawiasach po prawej stronie (czyli umieszczenie ich w *krotce*). Operator wyrażenia formatującego % oczekuje po prawej stronie pojedynczego elementu lub krotki elementów.

W trzecim przykładzie ponownie wstawiane są trzy wartości — liczba całkowita, liczba zmiennoprzecinkowa i listy, jednak warto zauważyć, że wszystkie miejsca docelowe po lewej stronie to znacznik %s, co oznacza konwersję na łańcuch znaków. Ponieważ każdy typ obiektu można przekształcić na łańcuch znaków (taki wykorzystywany w czasie wyświetlania), każdy typ obiektu zadziała w połączeniu z kodem konwersji %s. Z tego powodu, o ile nie wykonujemy jakiegoś formatowania specjalnego, %s jest często jedynym znacznikiem, który trzeba zapamiętać jako wyrażenie formatujące.

Należy również pamiętać, że formatowanie zawsze tworzy nowy łańcuch znaków i nie zmienia łańcucha znajdującego się po lewej stronie operatora. Ponieważ łańcuchy znaków są niemutowalne, musi to działać w taki sposób. Tak jak wcześniej, jeżeli chcesz zachować rezultat formatowania, powinieneś przypisać go do zmiennej.

Składnia zaawansowanych wyrażeń formatujących

W przypadku bardziej zaawansowanego formatowania określonych typów można w wyrażeniach formatujących skorzystać z kodów konwersji przedstawionych w tabeli 7.4, które wprowadza się po znaku % w łańcuchu formatującym. Programiści języka C rozpoznają zapewne większość tych kodów, ponieważ formatowanie łańcuchów znaków w Pythonie obsługuje wszystkie często używane kody formatów funkcji `sprintf` z języka C (jednak zwraca wynik, zamiast go wyświetlać, jak robi to funkcja `printf`). Niektóre z kodów z tabeli udostępniają alternatywne sposoby formatowania tego samego typu. Przykładowo `%e`, `%f` oraz `%g` zapewniają alternatywne sposoby formatowania liczb zmiennoprzecinkowych.

Tabela 7.4. Kody typów stosowane w wyrażeniach formatujących

Kod	Znaczenie
s	Łańcuch znaków (lub dowolny obiekt <code>str(X)</code>)
r	To samo co s, z tym że wykorzystuje funkcję <code>repr</code> , a nie <code>str</code>
c	Znak (<code>int</code> lub <code>str</code>)
d	Liczba dziesiętna
i	Liczba całkowita
u	To samo co d (przestarzałe, dawniej wymuszało liczbę całkowitą bez znaku)
o	Liczba ósemkowa (podstawa 8)
x	Liczba szesnastkowa (podstawa 16)
X	To samo co x, jednak wyświetlane wielkimi literami
e	Liczba zmiennoprzecinkowa w formacie wykładniczym, małą literą
E	To samo co e, wyświetlane wielką literą
f	Liczba zmiennoprzecinkowa w zapisie dziesiętnym
F	Liczba zmiennoprzecinkowa w zapisie dziesiętnym (wielkie litery)
g	Zmiennoprzecinkowe e lub f
G	Zmiennoprzecinkowe E lub F
%	Literal % (zapisywany jako %%)

Tak naprawdę znaczniki konwersji w formatowanym łańcuchu znaków po lewej stronie wyrażenia obsługują różne operacje konwersji za pomocą własnej składni. Ogólna struktura znacznika konwersji wygląda następująco:

`%[(nazwa)][opcje][szerokość][.precyzja]kod_typu`

Kody typów konwersji z tabeli 7.4 umieszcza się na końcu znacznika formatującego. Pomiędzy znakiem % a kodem typu można umieścić dodatkowe informacje formatujące, takie jak:

- Nazwa klucza do indeksowania słownika znajdującego się po prawej stronie wyrażenia.
- Opcje (flagi) określające na przykład wyrównywanie do lewej strony (-), znak liczby (+), spację przed liczbami dodatnimi i znak - przed liczbami ujemnymi czy dopełnienie zerami (0).

- Całkowita szerokość pola dla podstawionego tekstu.
- Liczba cyfr (*precyzja*) wyświetlana po przecinku dla liczb zmiennoprzecinkowych.

Zarówno szerokość pola, jak i *precyzja* mogą być zastąpione gwiazdką *, co będzie oznaczało, że powinny pobierać swoje wartości z kolejnych elementów krotki po prawej stronie wyrażenia formatującego (co jest bardzo przydatne, gdy nie znamy tych wartości do czasu uruchomienia i wykonania wyrażenia). Jeżeli nie potrzebujesz żadnego z tych dodatkowych narzędzi, to wystarczy, że użyjesz prostego znacznika %s, który zostanie zastąpiony odpowiadającą mu wartością z ciągu formatującego, niezależnie od jej typu.

Przykłady zaawansowanych wyrażeń formatujących

Składnia formatowaniałańcuchów znaków jest w pełni opisana w standardowej dokumentacji Pythona, jednak by zademonstrować przykład jej zastosowania, przyjrzyjmy się kilku przykładom. Poniższy kod najpierw formatuje liczbę całkowitą w sposób domyślny, a następnie umieszcza ją w polach sześcioczątkowych odpowiednio z wyrównaniem do lewej strony i z dopełnieniem zerami.

```
>>> x = 1234
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234 ...001234'
```

Znaczniki %e, %f oraz %g wyświetlają liczby zmiennoprzecinkowe na różne sposoby, co pokazano w poniższym przykładzie z interaktywnej sesji Pythona; znacznik %E spełnia tę samą rolę co %e, z jedną różnicą — litera e w zapisie wykładniczym jest wielką literą, natomiast znacznik %g dobiera format zapisu w zależności od wartości liczby (liczba będzie wyświetlana w zapisie wykładniczym, kiedy wartość wykładnika jest mniejsza niż -4 lub nie mniejsza niż precyzja liczby; w przeciwnym wypadku liczba jest wyświetlana w zapisie dziesiętnym f, z domyślną, 6-cyfrową precyzją):

```
>>> x = 1.23456789
>>> x                                     # w wersjach wcześniejszych niż 2.7 i 3.1
domyślnie wyświetlanych było więcej cyfr
1.23456789
>>> '%e | %f | %g' % (x, x, x)
'1.234568e+000 | 1.234568 | 1.23457'
>>> '%E' % x
'1.234568E+000'
```

W przypadku liczb zmiennoprzecinkowych można osiągnąć różne dodatkowe efekty w zakresie formatowania dzięki określeniu wyrównania do lewej strony, wypełnienia zerami, znaków liczb, całkowitej szerokości pola i liczby cyfr po miejscu dziesiętnym. W przypadku prostszych zadań można po prostu przekształcić je nałańcuchy znaków za pomocą wyrażenia formatującego lub zaprezentowanej wcześniej wbudowanej funkcji str.

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23 | 01.23 | +001.2'
>>> '%s' % x, str(x)
('1.23456789', '1.23456789')
```

Jeżeli szerokości pól przeznaczonych na poszczególne elementy formatowanego łańcucha znaków nie są znanne przed uruchomieniem programu, możesz użyć dynamicznie obliczanej szerokości i precyzji, oznaczając je za pomocą znaku * w ciągu formatującym, co spowoduje, że szerokość pola zostanie odczytana z krotki danych po prawej stronie operatora % — w poniższym przykładzie liczba 4 w krotce określa precyzję trzeciej liczby.

```
>>> '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

Jeżeli jesteś zainteresowany innymi możliwościami wyrażeń formatujących, powinieneś samodzielnie poeksperymentować z pokazanymi przykładami.

Wyrażenia formatujące oparte na słowniku

Otwiera to drzwi do korzystania z formatowania jako narzędzia szablonów. Do tej pory poznaliśmy tylko słowniki w rozdziale 4., ale oto przykład, który pokazuje podstawy:

Formatowanie łańcuchów znaków pozwala również, aby znaczniki konwersji znajdujące się po lewej stronie wyrażenia odwoływały się do kluczy w słowniku zdefiniowanym po prawej stronie i pobierały z niego odpowiednie wartości. Dzięki takiemu rozwiązaniu wyrażenia formatujące mogą być używane jako narzędzia szablonów. Do tej pory o słownikach wspominaliśmy tylko krótko w rozdziale 4., dlatego poniżej zamieszczamy przykład, który przedstawia sposób zastosowania słowników w wyrażeniach formatujących:

```
>>> "%(ilość)d %(twarz)s" % {"ilość":1, "twarz":"mielonka"}
'1 mielonka'
```

W powyższym kodzie klucze (ilość) i (twarz), znajdujące się po lewej stronie wyrażenia formatującego, odnoszą się do kluczy słownika znajdującego się po prawej stronie wyrażenia i służą do pobierania odpowiadających im wartości. Programy generujące tekst w formatach takich jak HTML czy XML często korzystają z tej techniki — można dzięki niej zbudować słownik wartości i zastąpić je wszystkie naraz za pomocą jednego wyrażenia formatującego, wykorzystującego referencje oparte na kluczach słownika (zwróć uwagę, że w kolejnym przykładzie pierwszy komentarz znajduje się nad wierszem zawierającym potrójnym cudzysłów, więc nie jest dodawany do ciągu znaków; przykład został wykonany w środowisku IDLE, które nie wyświetla monitu „...” dla kontynuacji wierszy):

```
>>>                                                 ## Szablon ze znacznikami
podstawiania
>>> reply = """
Witamy...
Witaj %(name)s!
Twój wiek to %(age)s lat.
"""

>>> values = {'name': 'Amadeusz', 'age': 40}      # Słownik wartości do
podstawienia
>>> print reply % values                         # Operacja podstawienia
Witamy...
Witaj Amadeusz!
Twój wiek to 40 lat.
```

Ta sama sztuczka jest również wykorzystywana w połączeniu z wbudowaną funkcją `vars`, która zwraca słownik zawierający wszystkie zmienne istniejące w miejscu jej wywołania.

```
>>> food = 'mielonka'  
>>> qty = 10  
>>> vars( )  
{'food': 'mielonka', 'qty': 10, ...plus wbudowane nazwy Pythona... }
```

Kiedy użyjemy funkcji `vars` po prawej stronie wyrażenia formatującego, w formatowanym łańcuchu można odwoływać się do zmiennych po ich nazwach, które spełniają w takiej sytuacji rolę kluczów słownika.

```
>>> "%(qty)d razy %(food)s" % vars( )          #Nazwy zmiennych są kluczami w  
słowniku vars()  
'10 razy mielonka'
```

Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 8. Przykłady konwersji łańcuchów znaków na ciągi znaków reprezentujących liczby szesnastkowe i ósemkowe za pomocą kodów formatujących `%x` i `%o` znajdują się w rozdziale 5. i nie będziemy ich tutaj powtarzać. Dodatkowe przykłady wyrażeń formatujących pojawiają się również w następnej i w ostatniej sekcji tego rozdziału, gdzie będziemy je porównywać do formatowania z użyciem wywołania metody `format`.

Formatowanie łańcuchów z użyciem metody `format`

Jak wspominaliśmy wcześniej, w Pythonie 2.6 i 3.0 został wprowadzony nowy sposób formatowania łańcuchów znaków, uznawany przez niektórych programistów za bardziej zgodny z duchem Pythona. W przeciwnieństwie do wyrażeń formatujących metody formatujące nie wykorzystują składni zapożyczonej z funkcji `printf` języka C i w założeniu miały być bardziej czytelne i jednoznaczne. Z drugiej strony, nowa technika wykorzystuje jednak pewne koncepcje zastosowane w funkcji `printf`, takie jak kody typów i specyfikacje formatowania. Co więcej, technika ta w znacznym stopniu dubluje możliwości wyrażeń formatujących (czasem nawet wymagając zastosowania większej ilości kodu do realizacji tych samych operacji) i w zaawansowanych zastosowaniach bywa nie mniej skomplikowana w użyciu. Z tego powodu trudno jednoznacznie polecić jedną z dostępnych opcji formatowania, dlatego programiści powinni po prostu zrozumieć podstawy każdej z nich i podjąć decyzję samodzielnie. Na szczęście oba sposoby są na tyle podobne, że wiele podstawowych koncepcji nakłada się na siebie.

Podstawy

Metoda `format` obiektów reprezentujących łańcuchy znaków, dostępna w Pythonie 2.6, 2.7 i 3.x, oparta jest na normalnej składni wywołania funkcji, a nie na wyrażeniu. Metoda ta wykorzystuje formatowany ciąg znaków jako szablon i akceptuje dowolną liczbę argumentów reprezentujących wartości do podstawienia zgodnie ze znacznikami umieszczonymi w szablonie.

Użycie metody `format` wymaga znajomości jej składni i sposobów wywołania, ale na szczęście nie jest to zbyt skomplikowane. W formatowanym łańcuchu znaków umieszczać możemy znaczniki podstawiania w postaci nawiasów klamrowych zawierających odwołanie do argumentu

zajdującego się na określonej pozycji (na przykład {1}) lub posiadającego określzoną nazwę (na przykład {food}). Jak się przekonasz podczas omawiania sposobów przekazywania argumentów w rozdziale 18., argumenty funkcji i metod mogą być przekazywane przez pozycję argumentu (tzw. argumenty pozycyjne) lub przez nazwę słowa kluczowego, a zdolność Pythona do przekazywania do funkcji lub metody dowolnej liczby argumentów pozwala na stosowanie takiego ogólnego wzorca wywoływanego tej metody. Na przykład:

```
>>> template = '{0}, {1} i {2}'                                # Podstawianie
pozycyjne

>>> template.format('Tytus', 'Romek', 'Atomek')
'Tytus, Romek i Atomek'

>>> template = '{pieczywo}, {wędлина} i {nabiał}'          # Podstawianie po
słowie kluczowym

>>> template.format(pieczywo='cheb', wędлина='szynka', nabiał='jajka')
'chleb, szynka i jajka'

>>> template = '{pieczywo}, {0} i {nabiał}'                  # Obydwie techniki

>>> template.format('szynka', pieczywo='chleb', nabiał='jajka')
'chleb, szynka i jajka'

>>> template = '{} , {} i {}'                                # Podstawianie
przez pozycję względną

>>> template.format('chleb', 'szynka', 'jajka')            # Nowość w wersjach
3.1 i 2.7

'chleb, szynka i jajka'
```

Dla porównania: wyrażenia formatujące z ostatniej sekcji mogą być nieco bardziej zwięzłe, ale używają słowników zamiast argumentów słów kluczowych i nie pozwalają na tak dużą elastyczność źródeł, z których pobierane są poszczególne wartości (co może być wadą albo zaletą w zależności od Twoich potrzeb); więcej na temat porównania tych obu technik znajdziesz w przykładzie poniżej:

```
>>> template = '%s, %s i %s'                                # To samo z wykorzystaniem
wyrażenia formatującego

>>> template % ('chleb', 'szynka', 'jajka')
'chleb, szynka i jajka'

>>> template = '%(pieczywo)s, %(wędлина)s i %(nabiał)s'
>>> template % dict(pieczywo='chleb', wędлина='szynka', nabiał='jajka')
'chleb, szynka i jajka'
```

Zwróć uwagę na użycie funkcji dict() do utworzenia słownika z argumentów, wprowadzonej w rozdziale 4. i omówionej w całości w rozdziale 8.; jest to często wygodna i bardziej zwięzła alternatywa dla literała {...}. Oczywiście formatowany łańcuch znaków w wywołaniu metody format może być również literałem, który tworzy ciąg tymczasowy, a w miejscach docelowych mogą być podstawiane dowolne typy obiektów, podobnie jak to było w wyrażeniach ze znacznikiem %s:

```
>>> '{pieczywo}, {0} i {nabiał}'.format(42, pieczywo=3.14, nabiał=[1, 2])
'3.14, 42 i [1, 2]'
```

Podobnie jak w przypadku wyrażeń formatujących z operatorem %, metoda `format` tworzy i zwraca nowy łańcuch znaków, który można wyświetlić lub przypisać do zmiennej w celu dalszego użycia (jak pamiętamy, łańcuchy znaków są niemutowalne, zatem metoda `format` musi tworzyć nowy obiekt znakowy). Formatowanie znaków jest stosowane nie tylko do wyświetlania:

```
>>> X = '{pieczywo}, {0} i {nabiał}'.format(42, pieczywo=3.14, nabiał=[1, 2])
>>> X
'3.14, 42 i [1, 2]'
>>> X.split(' i ')
['3.14', '42', '[1, 2]']
>>> Y = X.replace('i', 'ale pod żadnym pozorem nie')
>>> Y
'3.14, 42 ale pod żadnym pozorem nie [1, 2]'
```

Używanie kluczy, atrybutów i przesunięć

Podobnie jak wyrażenia formatujące z użyciem operatora %, metoda `format` obsługuje parametry formatowania pozwalające na obsługę bardziej zaawansowanych sytuacji. Na przykład szablony formatujące mogą wykorzystywać nazwy atrybutów i kluczy słowników. Jak w składni Pythona, nawiasy kwadratowe służą do określania nazw kluczy słownika, a nazwa po kropce określa nazwę atrybutu — wszystko to w ramach elementu podstawianego do szablonu po indeksie lub słowie kluczowym. Poniższy listing prezentuje kilka sposobów użycia tych możliwości. Pierwszy przykład wykorzystuje wartość klucza 'spam' przekazanego słownika oraz atrybut `platform` załadowanego wcześniej modułu `sys`. Drugi przykład wykonuje to samo zadanie, ale obiekty są podstawiane po słowie kluczowym zamiast po indeksie.

```
>>> import sys
>>> 'Mój {1[spam]} ma zainstalowany system {0.platform}'.format(sys, {'spam': 'laptop'})
'Mój laptop ma zainstalowany system win32'
>>> 'Mój {map[spam]} ma zainstalowany system {sys.platform}'.format(sys=sys,
map={'spam': 'laptop'})
'Mój laptop ma zainstalowany system win32'
```

Nawiasy kwadratowe w szablonach formatujących mogą również służyć do określania indeksów list (i innych sekwencji), ale działają tylko zwykle indeksy dodatnie, zatem ta możliwość nie jest tak ogólna, jak można by oczekwać. Podobnie jak w przypadku wyrażeń formatujących z operatorem %, w celu uzyskania dostępu do ujemnych indeksów lub wycinków albo użycia innych skomplikowanych wyrażeń należy wykonać je poza szablonem formatowania. W poniższym przykładzie warto zwrócić uwagę na użycie wyrażenia `*parts`, które służy do rozpakowania elementów krotki do indywidualnych argumentów funkcji, co zostało szczegółowo omówione w rozdziale 18.

```
>>> somelist = list('JAJKO')
>>> somelist
['J', 'A', 'J', 'K', 'O']
>>> 'pierwszy={0[0]}, trzeci={0[2]}'.format(somelist)
```

```

'pierwszy=J, trzeci=J'

>>> 'pierwszy={0}, ostatni={1}'.format(somelist[0], somelist[-1]) # [-1] w
szablonie formatującym nie działa
'pierwszy=J, ostatni=0'

>>> parts = somelist[0], somelist[-1], somelist[1:3]           # [1:3] w
szablonie nie zadziała

>>> 'pierwszy={0}, ostatni={1}, środkowy={2}'.format(*parts)    # Lub '{}' w
wersjach 2.7/3.1+
"pierwszy=J, ostatni=0, środkowy=[ 'A', 'J' ]"

```

Zaawansowana składnia wywołań metody format

Kolejne podobieństwo do wyrażeń formatujących polega na tym, że w szablonie można wykorzystać dodatkowe parametry pozwalające zmienić detale formatowania. W przypadku metod formatujących po identyfikacji pola podstawienia należy wpisać dwukropki, a następnie deklarację formatowania określającą rozmiar pola, wyrównanie i kod typu. Oto formalna składnia identyfikatora pola podstawienia w szablonie przy użyciu metod formatujących — wszystkie cztery elementy są opcjonalne i muszą pojawiać się bez spacji:

{nazwa_pola komponent !znacznik_przekształcenia :specyfikacja_formatu}

Znaczenie poszczególnych elementów:

- *nazwa_pola* jest opcjonalnym numerem lub nazwą słowa kluczowego argumentu; można ją pominąć i w zamian użyć względnej numeracji argumentów, która jest dostępna w wersjach 2.7, 3.1 i nowszych.
- *komponent* jest ciągiem składającym się z zera lub więcej odwołań do nazwy lub pozycji, używanym do pobierania atrybutów i wartości pozycji argumentów; można go pominąć, aby użyć całej wartości argumentu.
- *znacznik_przekształcenia*, jeżeli jest obecny, rozpoczyna się od znaku wykryznika (!), po którym następują kody r, s lub a, powodujące wywołanie odpowiednio funkcji repr, str lub ascii.
- *specyfikacja_formatu*, jeżeli jest obecna, rozpoczyna się od znaku dwukropka (:), po którym następuje ciąg znaków określający sposób prezentacji poszczególnych wartości; może zawierać takie szczegóły jak szerokość pola, wyrównanie, wypełnienie, precyzja wartości dziesiętnych itp.; może się kończyć opcjonalnym kodem typu danych.

Element *specyfikacja_formatu*, znajdujący się po znaku dwukropka, posiada swoją własną, rozbudowaną składnię, która została opisana poniżej (nawiasy klamrowe sygnalizują elementy opcjonalne):

[[wypełnienie]wyrównanie][znak][#][0][szerokość][,][.precyzja][kod_typu]

Wypełnienie może być dowolnym znakiem wypełniającym, z wyjątkiem{ i }; *wyrównanie* określa się jednym ze znaków: <, >, = lub ^ dla, odpowiednio, wyrównania do lewej, do prawej, dopełnienia po symbolu znaku liczby lub wycentrowania; *znakiem* może być +, - lub spacja; a opcja (,) powoduje ustawienie przecinka jako znaku separatora tysięcy (dostępne od wersji Python 2.7 i 3.1). *Szerokość* i *precyzja* są takie same jak w wyrażeniach formatujących %, a specyfikacja formatu może także zawierać zagnieżdżone znaczniki {} z nazwami pól, aby dynamicznie pobierać wartości z listy argumentów (podobnie jak znak * w wyrażeniach formatujących).

Kody typów prawie dokładnie pokrywają się z analogcznymi kodami dla wyrażeń formatujących przedstawionymi w tabeli 7.4; w przypadku metody format dodatkowo obsługiwany jest kod b

wykorzystywany do prezentowania liczb całkowitych w notacji dwójkowej (co stanowi odpowiednik funkcji wbudowanej `bin`). W szablonach formatujących metody `format` można bezpośrednio używać znaku `%` do wyświetlania wartości procentowych, a do formatowania liczb całkowitych w notacji dziesiętnej używany jest wyłącznie kod `d` (nie stosuje się i ani `u`). Zauważ, że w przeciwnieństwie do znacznika `%s` używanego w wyrażeniach formatujących zastosowanie tutaj kodu typu `s` wymaga argumentu w postaci obiektu reprezentującego ciąg znaków; aby zaakceptować dowolny typ, powinieneś po prostu pominąć kod typu.

Więcej szczegółowych informacji na temat składni podstawień znajdziesz w dokumentacji Pythona. Poza formatowaniem przy użyciu metody `format` pojedynczy obiekt może być również sformatowany za pomocą wbudowanej funkcji `format(obiekt, formatspec)`, której metoda `format` używa wewnętrznie, i może być dostosowany za pomocą przeciążenia operatora `__format__` w klasach zdefiniowanych przez użytkownika (zobacz część VI książki).

Przykłady zaawansowanego formatowania łańcuchów znaków z użyciem metody `format`

Jak widać, składnia formatowania łańcuchów tekstu może być dosyć złożona. Ponieważ w takich przypadkach najlepszym sojusznikiem jest praktyka, pokażemy teraz kilka przykładów. W pierwszym przykładzie poniżej, zapis `{0:10}` oznacza pierwszy argument pozycyjny, który będzie sformatowany w polu o szerokości 10 znaków, `{1:<10}` oznacza drugi argument pozycyjny, wyrównany do lewej strony w polu o szerokości 10 znaków, natomiast `{0.platform:>10}` oznacza atrybut `platform` tego pierwszego argumentu pozycyjnego, który będzie wyrównany do prawej w polu o szerokości 10 znaków (ponownie zwróć uwagę na sposób użycia funkcji `dict()`, przekształcającej argumenty ze słowami kluczowymi na słownik, o której wspominaliśmy już w rozdziale 4., a szczegółowo omówimy ją w rozdziale 8.):

```
>>> '{0:10} = {1:<10}'.format('jajo', 123.4567)          # W Pythonie 3.3
'jajo = 123.457'
>>> '{0:>10} = {1:<10}'.format('jajo', 123.4567)
' jajo = 123.457 '
>>> '{0.platform:>10} = {[kind]:<10}'.format(sys, dict(kind='laptop'))
' win32 = laptop '
```

W wersjach 2.7, 3.1 i nowszych Pythona możesz pominąć numery, jeżeli wybierasz argumenty od lewej do prawej z automatycznym numerowaniem względnym — chociaż takie rozwiązanie sprawia, że kod jest mniej czytelny i tym samym neguje jednouzgodność z wspomnianymi wcześniejszymi przewagami formatowania z użyciem metody `format` nad wyrażeniami formatującymi (patrz odpowiednia uwaga wcześniej):

```
>>> '{:10} = {:10}'.format('jajo', 123.4567)
'jajo = 123.4567'
>>> '{:>10} = {:<10}'.format('jajo', 123.4567)
' jajo = 123.4567 '
>>> '{.platform:>10} = {[kind]:<10}'.format(sys, dict(kind='laptop'))
' win32 = laptop '
```

Liczby zmiennoprzecinkowe obsługują te same kody typów i parametry formatowania co wyrażenia formatujące z operatorem `%`. Na przykład `{2:g}` oznacza trzeci argument sformatowany w domyślnej reprezentacji zmiennoprzecinkowej zgodnej z kodem `g`, natomiast

{1:.2f} spowoduje zastosowanie formatowania typu f z dwoma miejscami po przecinku, a {2:06.2f} spowoduje zastosowanie pola o szerokości sześciu znaków z wypełnieniem zerami od lewej strony:

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'
>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Metoda `format` obsługuje również formatowanie liczb całkowitych w notacjach szesnastkowej, ósemkowej i dwójkowej. W tym przypadku formatowanie jest odpowiednikiem użycia funkcji wbudowanych, odpowiednio: `hex`, `oct` i `bin`.

```
>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255)      # Hex, octal, binary
'FF, 377, 11111111'
>>> bin(255), int('11111111', 2), 0b11111111          # Przekształcenie z/do
notacji dwójkowej
('0b11111111', 255, 255)
>>> hex(255), int('FF', 16), 0xFF                      # Przekształcenie z/do
notacji ósemkowej
('0xff', 255, 255)
>>> oct(255), int('377', 8), 0o377                  # Przekształcenie z/do
notacji szesnastkowej, w wersji 3.x
('0o377', 255, 255)                                    # wersje 2.x akceptują i
wyświetlają tę liczbę w zapisie 0377
```

Parametry formatowania mogą być zakodowane w szablonie formatującym, mogą być też odczytywane z listy argumentów metody `format`. Służy do tego składnia zagnieżdżonych odwołań, która działa analogicznie do składni gwiazdki w wyrażeniach formatujących dla atrybutów szerokości i precyzji:

```
>>> '{0:.2f}'.format(1 / 3.0)                         # Parametry zakodowane w
szablonie
'0.33'
>>> '%.2f' % (1 / 3.0)                                # To samo dotyczy
wyrażenia
'0.33'
>>> '{0:{1}f}'.format(1 / 3.0, 4)                      # Wartość odczytywana z
argumentów
'0.3333'
>>> '%.*f' % (4, 1 / 3.0)                            # Podobnie w przypadku
wyrażeń
'0.3333'
```

W Pythonie w wersjach 2.6 i 3.0 została wprowadzona nowa funkcja `format`, która może być użyta do formatowania pojedynczego elementu. Ma ona być alternatywą dla stosowania metody `format`, a jej działanie jest analogiczne do prostych wyrażeń formatujących z operatorem `%`.

```
>>> '{0:.2f}'.format(1.2345) # Metoda łańcucha znaków
'1.23'
>>> format(1.2345, '.2f') # Funkcja wbudowana
'1.23'
>>> '%.2f' % 1.2345 # Wyrażenie
'1.23'
```

Technicznie rzecz biorąc, wbudowana funkcja `format` wywołuje metodę `__format__` obiektu, co metoda `str.format` wykonuje wewnętrznie dla każdego sformatowanego elementu. Jednak zarówno funkcja `format`, jak i metoda `format` wymagają większej ilości kodu niż wyrażenia formatujące %, o czym opowiemy już w kolejnym podrozdziale.

Porównanie metody format z wyrażeniami formatującymi

Jeżeli uważnie przestudiowałeś poprzedni podrozdział, z pewnością zauważyleś, że metoda `format` przypomina wyrażenia formatujące w zakresie referencji pozycyjnych i kluczy słownikowych, szczególnie w przypadku użycia zaawansowanych opcji formatowania. W rzeczywistości w typowych zastosowaniach wyrażenia formatujące mogą być łatwiejsze do zakodowania niż formatowanie z użyciem wywołania metody `format`, szczególnie gdy używasz ogólnych znaczników `%s`, a nawet automatycznego numerowania pól dodanych w wersjach 2.7 i 3.1:

```
print('%s=%s' % ('spam', 42))          # Wyrażenie formatujące; wszystkie  
wersje 2.x/3/x+  
  
print('{0}={1}'.format('spam', 42))      # Metoda format; wersje 3.0+ i 2.6+  
  
print('{0}={1}'.format('spam', 42))      # Z automatycznym numerowaniem; wersje  
3.1+ i 2.7
```

Jak się przekonasz już za chwilę, w bardziej skomplikowanych przypadkach takie różnice się wyrównują (złożone zadania są po prostu... bardziej złożone, niezależnie od użytej metody) i wielu programistów uważa metodę format za niepotrzebnie nadmiarową, biorąc pod uwagę wszechstronność wyrażeń formatujących.

Z drugiej strony, metoda `format` posiada również szereg potencjalnych zalet. Na przykład oryginalne wyrażenie z operatorem `%` nie potrafi obsługiwać słów kluczowych, odwołań do atrybutów i kodów binarnych, ale podobne rezultaty łatwo uzyskać, odpowiednio używając odwołań do kluczy słownika. Aby zaobserwować obszary pokrywania się tych dwóch technik formatowania, przeanalizujemy następujący listing prezentujący wyrażenia formatujące odpowiadające powyższym wywołaniom metody `format`.

```
>>> '%s, %s and %s' % (3.14, 42, [1, 2]) # Dowolne typy
'3.14, 42 and [1, 2]'

>>> 'Mój %(kind)s ma zainstalowany system %(platform)s' % {'kind': 'laptop',
'platform': sys.platform}
'Mój laptop ma zainstalowany system win32'

>>> 'Mój %(kind)s ma zainstalowany system %(platform)s' % dict(kind='laptop',
platform=sys.platform)
```

```
'Mój laptop ma zainstalowany system win32'
>>> somelist = list('JAJKA')
>>> parts = somelist[0], somelist[-1], somelist[1:3]
>>> 'first=%s, last=%s, middle=%s' % parts
"pierwsza=J, ostatnia=0, środkowe=['A', 'J']"
```

W przypadku bardziej skomplikowanego formatowania obie techniki są zbliżone pod względem stopnia złożoności kodu, ale porównując poniższy listing z analogicznym kodem wykorzystującym metodę `format`, musimy stwierdzić, że wyrażenia formatujące z operatorem `%` są prostsze i bardziej czytelne; w Pythonie 3.3 wyglądają tak:

```
# Zastosowanie specyficznego formatowania
>>> '%-10s = %10s' % ('jajo', 123.4567)
'jajo = 123.4567'
>>> '%10s = %-10s' % ('jajo', 123.4567)
' jajo = 123.4567 '
>>> '%(plat)10s = %(kind)-10s' % dict(plat=sys.platform, kind='laptop')
' win32 = laptop '
# Liczby zmiennoprzecinkowe
>>> '%e, %.3e, %g' % (3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
# Notacja szesnastkowa i ósemkowa, ale brak dwójkowej (zobacz dalej)
>>> '%x, %o' % (255, 255)
'ff, 377'
```

Metoda `format` obsługuje kilka zaawansowanych możliwości, których nie potrafią obsłużyć wyrażenia formatujące, ale nawet bardziej zaawansowane formatowania wyglądają praktycznie tak samo skomplikowanie w każdej z technik formatujących. Na przykład poniższy listing w każdej z zastosowanych technik daje te same wyniki, z polami o ustalonej szerokości i określonym wyrównaniem, z zastosowaniem różnych metod dostępu do argumentów.

```
# Obie metody wykorzystują zakodowane odwołania
>>> import sys
>>> 'Mój {1[kind]}: <8> ma zainstalowany system {0.platform:>8}'.format(sys,
{'kind': 'laptop'})
'Mój laptop ma zainstalowany system win32'
>>> 'Mój %(kind)-8s ma zainstalowany system win32 %(plat)8s' %
dict(kind='laptop', plat=sys.platform)
'Mój laptop ma zainstalowany system win32'
```

W praktyce w programach rzadko stosuje się odwołania zakodowane w sposób statyczny, częściej tworzony jest kod budujący zestaw podstawień (na przykład aby zebrać dane wejściowe z formularza lub bazy danych i następnie podstawić je do szablonu HTML). Jeżeli weźmiemy pod uwagę tego typu wzorce zastosowań, podobieństwo między metodą format a wyrażeniami formatującymi z operatorem % okaże się jeszcze bardziej uderzające:

```
# Budowanie danych i dynamiczne formatowanie
>>> data = dict(platform=sys.platform, kind='laptop')
>>> 'Mój {kind:<8} ma zainstalowany system {platform:>8}'.format(**data)
'Mój laptop ma zainstalowany system win32'
>>> 'Mój %(kind)-8s ma zainstalowany system %(platform)8s' % data
'Mój laptop ma zainstalowany system win32'
```

Jak się przekonasz w rozdziale 18., argument **data w wywołaniu metody jest przykładem specjalnej składni, która rozpakowuje słownik kluczy i wartości na poszczególne argumenty w postaci par słów kluczowych „nazwa = wartość”, dzięki czemu można się do nich odwoływać według nazwy w wyrażeniu formatującym — jest to kolejne, nieuniknione odwołanie do nieomawianych jeszcze, zaawansowanych sposobów wywoływania funkcji, które może być traktowane jako kolejny minus metody format, szczególnie dla nowych użytkowników.



Rozszerzenia metody format w Pythonie 3.1 i 2.7: w Pythonie w wersji 3.1 oraz 2.7 wprowadzono do metody format obsługę separatorów tysięcy dla liczb, które wstawiane są pomiędzy kolejnymi trzycyfrowymi grupami cyfr. Aby tej opcji skorzystać, wystarczy dodać przecinek przed kodem typu oraz pomiędzy szerokością pola i precyzją, jeżeli występuje:

```
>>> '{0:d}'.format(999999999999)
'999999999999'
>>> '{0:,d}'.format(999999999999)
'999,999,999,999'
```

Wymienione wyżej wersje Pythona automatycznie przypisują odpowiednią numerację względną do kolejnych argumentów metody, o ile ich pozycje nie zostały zadeklarowane jawnie w formatowanym łańcuchu. Użycie takiego rozszerzenia nie ma jednak zastosowania we wszystkich przypadkach i może negować jedną z głównych zalet używania metody format — bardziej czytelny kod:

```
>>> '{:,d}'.format(999999999999)
'999,999,999,999'
>>> '{:,d} {:,d}'.format(99999999, 8888888)
'9,999,999 8,888,888'
>>> '{:.2f}'.format(296999.2567)
'296,999.26'
```

Więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji Pythona 3.1; możesz również zajrzeć do przykładów wstawiania przecinków i formatowania wartości walutowych omawianych w rozdziale 25. (skrypt o nazwie *formats.py*), aby zapoznać się z prostym rozwiązaniem, które można zimportować, a następnie użyć go w Pythonie w wersjach wcześniejszych niż

3.1 i 2.7. Jak to zwykle bywa w programowaniu, wdrożenie nowej funkcjonalności w dostosowanej do własnych potrzeb funkcji wielokrotnego użytku jest znacznie łatwiejsze niż poleganie na stałym zestawie wbudowanych narzędzi:

```
>>> from formats import commas, money  
>>> '%s' % commas(999999999999)  
'999,999,999,999'  
>>> '%s %s' % (commas(9999999), commas(8888888))  
'9,999,999 8,888,888'  
>>> '%s' % money(296999.2567)  
'$296,999.26'
```

Jak zwykle taką prostą funkcję można zastosować również w bardziej zaawansowanych kontekstach, takich jak narzędzia iteracyjne, które poznaliśmy w rozdziale 4. i które w pełni zbadamy w późniejszych rozdziałach:

```
>>> [commas(x) for x in (9999999, 8888888)]  
['9,999,999', '8,888,888']  
>>> '%s %s' % tuple(commas(x) for x in (9999999, 8888888))  
'9,999,999 8,888,888'  
>>> ''.join(commas(x) for x in (9999999, 8888888))  
'9,999,9998,888,888'
```

Nie zmienia to jednak w niczym faktu, że programiści Pythona często wolą tworzyć narzędzia dedykowane do konkretnego zastosowania niż korzystać z ogólnych technik programowania — jest to często kompromis, do którego powrócimy w następnej sekcji.

Jak zwykle to społeczność Pythona zdecyduje, czy próbę czasu przetrwają wyrażenia formatujące z operatorem %, czy metoda format, czy też wykorzystywane będą obie techniki. Aby lepiej poznać ich możliwości, powinieneś samodzielnie poeksperymentować z tymi mechanizmami oraz zapoznać się z dokumentacją używanej wersji Pythona.

Dlaczego miałbyś korzystać z metody format

Skoro już poświęciłem sporo miejsca na porównanie dwóch dostępnych technik formatowania łańcuchów znaków, powiniensem wyjaśnić, dlaczego mimo wszystko powinieneś rozważyć możliwość korzystania z metody format. Mówiąc w skrócie, choć metoda format wymaga czasem napisania większej ilości kodu, posiada również kilka niewątpliwych zalet:

- ma kilka dodatkowych możliwości, których nie mają wyrażenia formatujące z operatorem % (choć można to w taki czy inny sposób obejść);
- ma bardziej elastyczną składnię odwołań do wartości (chociaż może to trochę naciągany argument, a poza tym wyrażenia formatujące często mają odpowiedniki takich odwołań);
- podstawiane wartości mogą być definiowane w sposób bardziej jawnym (choć jest to opcjonalne);
- zamiast symbolu operatora używa bardziej oczywistej dla użytkownika nazwy metody (ale to z kolei wymaga większej ilości wpisywanych znaków w kodzie);

- nie wymaga stosowania osobnej składni dla pojedynczych i wielokrotnych podstawień (choć praktyka wskazuje, że jest to trywialne);
- jako funkcja może być używana w miejscach, w których nie możemy użyć wyrażenia (chociaż funkcje jednowierszowe zdają się temu zaprzeczać).

Choć obie opisane techniki są dziś dostępne, a wyrażenia formatujące są nadal szeroko stosowane, zastosowanie metody `format` staje się coraz bardziej popularne i w przyszłości może zyskać więcej uwagi deweloperów języka Python. Co więcej, ponieważ zarówno wyrażenia formatujące, jak i wywołania metody `format` mogą się pojawiać w każdym napotkanym kodzie, powinieneś dobrze rozumieć zasady działania *obu* tych rozwiązań. Skoro jednak to do Ciebie należy ostateczny wybór rozwiązań, jakiego będziesz używał w swoich programach, na zakończenie krótko omówmy jeszcze kilka zagadnień związanych ze stosowaniem obu technik.

Dodatkowe możliwości: wbudowane funkcje czy ogólne techniki programowania

Metoda `format` oferuje kilka możliwości, które w wyrażeniach formatujących nie są dostępne, takich jak formatowanie liczb w notacji dwójkowej czy separatory tysięcy (dostępne w wersjach 2.7, 3.1 i nowszych). Ale jak mieliśmy okazję się przekonać, korzystając z wyrażeń formatujących, zwykle możemy osiągnąć takie same efekty w nieco inny sposób. Oto przykład *formatowania liczb binarnych*:

```
>>> '{0:b}'.format((2 ** 16) -1)                                # Wyrażenia nie obsługują
formatowania liczb binarnych
'1111111111111111'
>>> '%b' % ((2 ** 16) -1)
ValueError: unsupported format character 'b' ...
>>> bin((2 ** 16) -1)                                         # ale inne, bardziej ogólne
sposoby działają całkiem dobrze
'0b1111111111111111'
>>> '%s' % bin((2 ** 16) - 1)                                    # działa zarówno w metodzie
format, jak i wyrażeniach formatujących
'0b1111111111111111'
>>> '{}'.format(bin((2 ** 16) - 1))                            # w wersjach 2.7/3.1+ względne
numerowanie argumentów
'0b1111111111111111'
>>> '%s' % bin((2 ** 16) - 1)[2:]                               # Odcinamy początkowe znaki
0b, aby otrzymać dokładną wartość binarną
'1111111111111111'
```

W podobny sposób odpowiednie użycie funkcji ogólnych może zastąpić opcję wstawiania *separatatora tysięcy*, dostępną w metodzie `format`, i pełniej wspierać dostosowywanie kodu do własnych potrzeb. W tym przypadku wystarczy prosta, kilkuwierszowa funkcja wielokrotnego użytku, która zapewni nam taką samą funkcjonalność bez konieczności stosowania dodatkowej składni dla specjalnych przypadków:

```
>>> '{:,d}'.format(999999999999)                                # Nowa funkcjonalność metody
format, dostępna w wersjach 3.1/2.7+
'999,999,999,999'
```


'Adam dev Adam'

Szczerze mówiąc, metoda `format` ma jeszcze bardziej rozbudowaną składnię podstawień do specjalnych zastosowań, co dla niektórych użytkowników może być kolejnym argumentem za używaniem tej metody, ale biorąc pod uwagę nakładanie się funkcjonalności i dodatkową złożoność składni, nowa użyteczność metody `format` wydaje się być niewielka, a jej dodatkowe możliwości są implementowane trochę na siłę w poszukiwaniu możliwych przypadków zastosowania. Moim skromnym zdaniem konieczność wprowadzania takich nadmiarowych metod stanowiących dodatkowe obciążenie pojęciowe dla programistów języka Python, którzy muszą teraz poznawać *oba* narzędzia, nie wydaje się wyraźnie uzasadniona.

Jawne odwołania do wartości: teraz opcjonalne i prawdopodobnie nie będą używane

Jednym z niewielu przypadków, w których metoda `format` wydaje się doskonalsza, jest podstawianie większej liczby wartości do szablonu formatującego. Przedstawiony w rozdziale 31. program *lister.py* wykorzystuje podstawianie sześciu elementów do jednego szablonu i w tym przypadku odwołanie pozycyjne typu `{i}` wydaje się dawać nieco bardziej czytelny kod w porównaniu do podstawiania z użyciem znacznika `%s`:

```
'\n%s<Klasa %s, adres %s:\n%s%s%s>\n' % (...) # Wyrażenie  
'\n{0}<Klasa {1}, adres {2}:\n{3}{4}{5}>\n'.format(...) # Metoda
```

Z drugiej strony, wykorzystując podstawienia z użyciem słownika w wyrażeniach `%`, można w znaczącym stopniu zniwelować tę różnicę. Trzeba też pamiętać, że powyższy przykład należy do najbardziej skrajnych przypadków komplikacji szablonów formatujących i nie jest powszechnie stosowany w praktyce. Bardziej typowe przypadki użycia są znacznie prostsze. Co więcej, w Pythonie 3.1 i 2.7 numerowanie odwołań do argumentów metody `format` w szablonie formatującym staje się opcjonalne, co sugeruje, że potencjalne zwiększenie czytelności kodu w takiej sytuacji staje się dosyć iluzoryczne.

```
>>> 'Zawsze {0} na {1} z {2}'.format('patrz', 'życie', 'humorem') # Python  
3.x, 2.6+  
  
'Zawsze patrz na życie z humorem'  
  
>>>  
  
>>> 'Zawsze {} na {} z {}'.format('patrz', 'życie', 'humorem') # Python  
3.1+, 2.7+  
  
'Zawsze patrz na życie z humorem'  
  
>>>  
  
>>> 'Zawsze %s na %s z %s' % ('patrz', 'życie', 'humorem') #  
Wszystkie wersje Pythona  
  
'Zawsze patrz na życie z humorem'
```

Biorąc pod uwagę zwięzłość kodu, drugi z przykładów jest prawdopodobnie lepszy niż pierwszy, ale wydaje się, że neguje największą zaletę korzystania z metody `format`. Porównajmy zatem przykłady formatowania liczb zmiennoprzecinkowych — tutaj wykorzystanie wyrażenia formatującego pozwala uzyskać jeszcze bardziej zwięzły i bardziej czytelny kod:

```
>>> '{0:f}, {1:.2f}, {2:05.2f}'.format(3.14159, 3.14159, 3.14159)  
'3.141590, 3.14, 03.14'  
  
>>> '{:f}, {:.2f}, {:06.2f}'.format(3.14159, 3.14159, 3.14159)  
'3.141590, 3.14, 003.14'
```

```
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Nazwy metod i argumenty neutralne kontekstowo — estetyka kodu kontra zastosowania praktyczne

Metoda `format` ma również tę zaletę, że zastępuje operator `%` bardziej mnemoniczną nazwą `format` i nie ma rozgraniczenia składowi między podstawianiem jednej i większej liczby wartości. Pierwsza z wymienionych zalet może nieco ułatwiać analizowanie kodu zwłaszcza początkującym użytkownikom (słowo kluczowe `format` jest tutaj bardziej oczywiste niż wiele tajemniczych znaków `%`), choć zdania są tutaj dosyć podzielone i ogólna korzyść wydaje się być niewielka.

Druga zaleta metody `format` jest już bardziej istotna: w przypadku wyrażeń formatujących podstawienie *pojedynczej* wartości pozwala podać ją bezpośrednio po operatorze `%`, ale w przypadku *więcej liczb* elementów musimy użyć krotki.

```
>>> '%.2f' % 1.2345          # Pojedyncza wartość
'1.23'
>>> '%.2f %s' % (1.2345, 99)    # Krotka z wieloma wartościami
'1.23 99'
```

Z technicznego punktu widzenia wyrażenia formatujące akceptują pojedynczą wartość do podstawienia *lub* krotkę zawierającą większą liczbę wartości. W związku z tym, ponieważ pojedynczy element może być podany samodzielnie lub w krotce, to jeżeli elementem podstawianym również jest krotka, musimy ją zapisać jako krotkę zagnieżdzoną — jest to być może rzadki, ale całkiem możliwy przypadek:

```
>>> '%s' % 1.23          # Pojedyncza wartość zapisana wprost
'1.23'
>>> '%s' % (1.23,)        # Pojedyncza wartość zapisana w krotce
'1.23'
>>> '%s' % ((1.23,),)      # Pojedyncza wartość będąca krotką i
zapisana w krotce
'(1.23,)'
```

Z drugiej strony metoda `format` ujednolica składnię, akceptując tylko ogólne argumenty funkcji w obu przypadkach, zamiast wymagać krotki dla wielu wartości lub pojedynczej wartości, która jest krotką:

```
>>> '{0:.2f}'.format(1.2345)      # Pojedyncza wartość
'1.23'
>>> '{0:.2f} {1}'.format(1.2345, 99)  # Wiele wartości
'1.23 99'
>>> '{0}'.format(1.23)            # Pojedyncza wartość zapisana wprost
'1.23'
>>> '{0}'.format((1.23,))
#Pojedyncza wartość będąca krotką
'(1.23,)'
```

Jeśli zawsze zatrzymujesz wartości w krotce i zignorujesz nieskorygowaną opcję, wyrażenie jest zasadniczo takie samo jak wywołanie metody tutaj. Takie podejście może być bardziej oczywiste dla początkujących użytkowników i pozytywnie wpływać na zmniejszenie liczby popełnianych błędów programistycznych. Wydaje się to być jednak niezbyt istotną kwestią — jeżeli nabierzesz nawyku ujmowania operandów wyrażenia % w krotkę, nawet dla pojedynczych podstawień, uzyskasz tę samą spójność składniową co w przypadku wywoływania metody `format`. Co więcej, zastosowanie metody `format` generuje dodatkowy koszt w postaci zwiększenia ilości kodu, jaki musi wprowadzić programista dla uzyskania tego samego efektu. Biorąc pod uwagę powszechnie stosowanie wyrażeń formatujących w historii Pythona, ten problem może być bardziej teoretyczny niż praktyczny i może nie uzasadniać konieczności przenoszenia istniejącego kodu do nowego narzędzia, które jest tak podobne do istniejących, sprawdzonych i skutecznych rozwiązań.

Funkcje a wyrażenia: niewielka wygoda

Ostateczne uzasadnienie istnienia metody `format` jest następujące: jest to *funkcja*, która będzie używana tam, gdzie wyrażenie formatujące pojawić się nie może, a to wymaga większej ilości informacji o funkcjach, niż udało się do tej pory przekazać, zatem nie będziemy się nad tym teraz rozwodzić. Wystarczy powiedzieć, że zarówno metoda `str.format`, jak i wbudowana funkcja `format` mogą być przekazywane do innych funkcji przechowywanych w innych obiektach i tak dalej. Wyrażenie takie jak % nie może być przekazywane bezpośrednio, ale nie jest to przecież zbytnie utrudnienie — opakowanie dowolnego wyrażenia w jednowierszową funkcję `def` czy `lambda` jest zadaniem trywialnym i pozwala na przekształcenie wyrażenia formatującego w funkcję o tych samych właściwościach (choć znalezienie powodu, aby tak postępować, może stanowić większe wyzwanie):

```
def myformat(fmt, args): return fmt % args          # Zobacz część IV książki  
myformat('%s %s', (88, 99))                         # Wywołanie własnej funkcji  
str.format('{} {}'.format, 88, 99)                   # Wywołanie funkcji  
wbudowanej  
  
otherfunction(myformat)                             # Twoja funkcja również  
jest obiektem
```

Tak czy owak, może to nie być żaden wybór. Chociaż wyrażenia formatujące są nadal powszechnie używane w kodzie Pythona, to obecnie możemy korzystać zarówno z tych wyrażeń, jak i metody `format`, a większość programistów pracując nad swoimi projektami z pewnością skorzysta na znajomości obu technik. Z pewnością opanowanie takich dublujących się rozwiązań będzie wymagała od początkujących programistów nieco większego nakładu pracy, ale na tym całym, ogromnym bazarze pomysłów, który nazywamy światem oprogramowania open source, zawsze będzie miejsce na coś więcej [3].



I jeszcze jedno: technicznie rzecz biorąc, w Pythonie są wbudowane trzy (a nie dwa) narzędzia do formatowania, jeżeli uwzględnimy wspomniane wcześniej szablony `template` z modułu `string`. Teraz gdy poznajesz już pozostałe dwa, możesz pokazać, jak to wygląda. Wyrażenia formatujące i metoda `format` również mogą być używane jako narzędzia szablonów przy odwoływaniu się do wartości podstawienia według nazw za pomocą kluczów słownika lub argumentów słów kluczowych:

```
>>> '%(num)i = %(title)s' % dict(num=7, title='Strings')  
'7 = Strings'  
>>> '{num:d} = {title:s}'.format(num=7, title='Strings')  
'7 = Strings'
```

```
>>> '{num} = {title}'.format(**dict(num=7, title='Strings'))  
'7 = Strings'
```

System szablonów modułu `string` także umożliwia odwoływanie się do wartości według nazwy poprzedzonej znakiem \$ jako kluczy słownika lub słów kluczowych, ale nie obsługuje wszystkich narzędzi pozostałych dwóch metod — jest to ograniczenie, które jednocześnie zapewnia prostotę będącą główną motywacją do używania tego narzędzia:

```
>>> import string  
>>> t = string.Template('$num = $title')  
>>> t.substitute({'num': 7, 'title': 'Strings'})  
'7 = Strings'  
>>> t.substitute(num=7, title='Strings')  
'7 = Strings'  
>>> t.substitute(dict(num=7, title='Strings'))  
'7 = Strings'
```

Więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji Pythona. Możliwe, że w praktyce spotkasz takie rozwiązanie (a także inne sposoby formatowania, używane w niezależnych modułach innych deweloperów) w kodzie programów napisanych w Pythonie; na szczęście ta technika jest bardzo prosta i zbyt rzadko spotykana, aby ją tutaj szerzej omawiać. Najlepszym rozwiązaniem dla większości początkujących użytkowników Pythona jest obecnie poznanie i używanie wyrażeń formatujących, metody `str.format` lub obu tych sposobów jednocześnie.

Generalne kategorie typów

Po omówieniu pierwszych obiektów kolekcji Pythona, czyli łańcuchów znaków możemy zatrzymać się jeszcze na chwilę w celu zdefiniowania kilku ogólnych koncepcji, które będą miały zastosowanie do wszystkich typów, z jakimi od teraz będziemy się spotykać. Jeżeli chodzi o typy wbudowane, okazuje się, że pewne operacje działają tak samo dla wszystkich typów z jednej kategorii, dlatego większość koncepcji wystarczy przedstawić tylko raz. Dotychczas omówiliśmy jedynie liczby i łańcuchy znaków, jednak ponieważ są one reprezentatywne dla dwóch z trzech ważnych kategorii typów w Pythonie, już teraz wiesz o innych typach więcej, niż Ci się wydaje.

Typy z jednej kategorii współdzielą zbiory operacji

Jak przekonałeś się już wcześniej, łańcuchy znaków to sekwencje niemutowalne — nie można ich zmieniać w miejscu (stąd *niemutowalne*) i są kolekcjami uporządkowanymi według pozycji, do których dostęp odbywa się z użyciem wartości przesunięcia (stąd *sekwencje*). Tak się składa, że na wszystkich sekwencjach omawianych w tej części książki można wykonywać te same operacje, które zostały zaprezentowane w tym rozdziale dla łańcuchów znaków. Dotyczy to na przykład konkatenacji, indeksowania czy wykonywania iteracji. Z bardziej formalnego punktu widzenia w Pythonie istnieją trzy kategorie typów danych (i powiązanych z nimi operacji), które mają taką ogólną naturę:

Liczby (całkowite, zmiennoprzecinkowe, dziesiętne, ułamki, inne)

Obsługują dodawanie, mnożenie i tym podobne.

Sekwencje (łańcuchy znaków, listy, krotki)

Obsługują indeksowanie, wycinki, konkatenację i tym podobne.

Odwzorowania (słowniki)

Obsługują indeksowanie po kluczu i tym podobne.

Pod ogólną etykietą „łańcuchy znaków” uwzględniamy tutaj również ciągi bajtowe z Pythona 3.x i ciągi znaków Unicode w Pythonie 2.x, o których wspomniałem na początku tego rozdziału (zobacz rozdział 37.). Zbiory same w sobie stanowią osobną, specyficzną kategorię danych (nie odwzorowują kluczy na wartości i nie są sekwencjami zachowującymi kolejność elementów), ponadto na razie nie omówiliśmy jeszcze odwzorowań (do słowników przejdziemy w kolejnym rozdziale). Pozostałe typy, z jakimi się spotkamy, będą jednak do siebie dość podobne. Na przykład dla dowolnych dwóch obiektów sekwencji X i Y:

- X + Y tworzy nowy obiekt sekwencji składający się z zawartości obu sekwencji będących operandami,
- X * N tworzy nowy obiekt sekwencji, zawierający N kopii operandu X.

Innymi słowy, operacje te działają tak samo na każdym rodzaju sekwencji, w tym na łańcuchach znaków, listach, krotkach i niektórych typach obiektów zdefiniowanych przez użytkownika. Jedyną różnicą jest to, że nowy obiekt wynikowy jest tego samego typu co operandy X i Y — po połączeniu dwóch list otrzymujemy nową listę, a nie łańcuch znaków. Indeksowanie, wycinki i inne operacje na sekwencjach działają w ten sam sposób na wszystkich sekwencjach; rodzaj przetwarzanego obiektu informuje Pythona, jakie zadanie ma wykonać.

Typy mutowalne można modyfikować w miejscu

Klasyfikacja typów na mutowalne i niemutowalne jest ważnym ograniczeniem, o którym należy pamiętać, ponieważ nowi użytkownicy często mają z nią problemy. Jeżeli obiekt jest typu niemutowalnego, nie można modyfikować jego wartości w miejscu. Kiedy spróbujemy to zrobić, Python zwróci błąd. Zamiast tego konieczne jest wykonanie kodu tworzącego nowy obiekt z nową wartością. Do podstawowych typów danych w Pythonie zaliczamy:

Typy niemutowalne (liczby, łańcuchy znaków, krotki, zamrożone zbiory)

Żaden z typów obiektów niemutowalnych nie obsługuje zmian w miejscu, chociaż zawsze możemy używać wyrażeń przekształcających obiekty w inne, nowe obiekty i przypisywać ich wyniki do zmiennych.

Typy mutowalne (listy, słowniki, zbiory, ciągi bytarray)

Typy mutowalne mogą być modyfikowane w miejscu z użyciem operacji, które nie tworzą nowych obiektów. Takie obiekty można kopować, tworząc nowe obiekty, jednak modyfikowanie w miejscu daje możliwość bezpośredniej modyfikacji istniejących danych.

Zasadniczo typy niemutowalne zapewniają pewien stopień integralności danych, gwarantując, że obiekt nie zostanie zmieniony przez inną część programu. Aby przypomnieć sobie, dlaczego to ma znaczenie, powinieneś zatrzymać się na chwilę i przypomnieć sobie, co oznacza modyfikowanie w miejscu. W kolejnym rozdziale dowiesz się, w jaki sposób listy, słowniki i krotki biorą udział w określaniu kategorii typów.

Podsumowanie rozdziału

W niniejszym rozdziale dogłębnie zapoznaliśmy się z obiektami reprezentującymi łańcuchy znaków. Dowiedzieliśmy się, czym są literaly łańcuchów znaków, i omówiliśmy operacje na tym typie danych, w tym wyrażenia działające na sekwencjach, formatowanie łańcuchów, wywoływanie ich metod oraz formatowanie łańcuchów znaków z użyciem wyrażeń % i metody format. Przy okazji omówiliśmy szczegółowo wiele różnych koncepcji, takich jak wycinki, wywołania metod i bloki łańcuchów znaków umieszczone w potrójnych apostrofach lub cudzysłówach. Zdefiniowaliśmy również pewne podstawowe dla różnych typów koncepcje. Sekwencje wspólnie dzielą na przykład cały zbiór operacji.

W kolejnym rozdziale będziemy kontynuować omawianie typów obiektów Pythona, przechodząc do najbardziej uniwersalnych kolekcji obiektów tego języka, czyli list i słowników. Jak się niebawem okaże, wiele z informacji, które uzyskaliśmy tutaj, będzie miało zastosowanie również do tych typów. Jak wspomniałem wcześniej, w ostatniej części książki wróćmy do pythonowego modelu łańcuchów znaków przy okazji analizy użycia tekstów i danych binarnych w formacie Unicode, które są bardzo użyteczne dla niektórych programistów. Najpierw jednak czas na krótki quiz podsumowujący niniejszy rozdział.

Sprawdź swoją wiedzę — quiz

1. Czy metody `find` łańcucha znaków można użyć do przeszukiwania listy?
2. Czy wyrażenie wycinające łańcuch znaków może zostać zastosowane do listy?
3. W jaki sposób można przekonwertować znak do postaci jego kodu ASCII? W jaki sposób można wykonać odwrotną operację, konwertując kod ASCII na znak?
4. W jaki sposób można w Pythonie zmienić łańcuch znaków?
5. Masz dany łańcuch znaków S o zawartości "j ,aj ,a". Podaj dwa sposoby wyodrębnienia z niego dwóch środkowych znaków.
6. Ile znaków znajduje się w łańcuchu "a\nb\x1f\000d"?
7. Do czego można by użyć modułu `string` zamiast wywołania metod łańcuchów znaków?

Sprawdź swoją wiedzę — odpowiedzi

1. Nie, ponieważ metody są zawsze specyficzne dla typu, co oznacza, że działają one wyłącznie na jednym typie danych. Wyrażenia typu X+Y i funkcje wbudowane, jak `len(X)`, są jednak uniwersalne i mogą działać na wielu typach obiektów. W tym przypadku na przykład wyrażenie testu przynależności `in` ma podobne działanie jak metoda `find` łańcucha znaków, ale może być użyte do przeszukiwania łańcuchów znaków oraz list. W Pythonie 3.x podjęto próbę pogrupowania metod według kategorii (na przykład zmienne typy `list` i `bytearray` posiadają podobne zbiory metod), jednak metody są bardziej związane z typami niż operatory.
2. Tak. W przeciwieństwie do metod wyrażenia są uniwersalne i mają zastosowanie do wielu typów. W tym przypadku wyrażenie z wycinkiem jest tak naprawdę operacją

na sekwencjach — działa na każdym typie obiektu sekwencji, w tym na łańcuchach znaków, listach oraz krotkach. Jedyną różnicą jest to, że kiedy tworzy się wycinek listy, z powrotem otrzymuje się listę.

3. Wbudowana funkcja `ord(S)` konwertuje jednoznakowy łańcuch na kod liczbowy znaku. Funkcja `chr(I)` przekształca z kolei kod liczbowy z powrotem na łańcuch znaków. Powinieneś jednak pamiętać, że takie liczby całkowite są tylko kodami znaków dla podstawowego zestawu znaków ASCII. W modelu Unicode ciągi tekstowe są tak naprawdę sekwencjami liczb całkowitych reprezentującymi kody znaków Unicode, które mogą wykraczać poza 7-bitowy zakres liczb zarezerwowany dla zestawu ASCII (więcej szczegółowych informacji o formacie Unicode znajdziesz w rozdziale 4. i rozdziale 37.).
4. Łańcuchy znaków są niemutowalne, zatem nie można ich modyfikować w miejscu. Można jednak uzyskać podobny efekt, tworząc nowy łańcuch — za pomocą konkatenacji, wycinka, wykonania wyrażenia formatującego czy użycia metody, takiej jak `replace` — a następnie przypisując wynik z powrotem do oryginalnej nazwy zmiennej.
5. Można utworzyć wycinek z tego łańcucha znaków za pomocą wyrażenia `S[2:4]` lub podzielić łańcuch w miejscu wystąpienia przecinka i zindeksować go za pomocą `S.split(',') [1]`. Oba rozwiązania warto wypróbować w sesji interaktywnej Pythona.
6. Sześć. Łańcuch znaków "a\nb\x1f\000d" zawiera następujące bajty: a, nowy wiersz (\n), b, binarne 31 (w postaci szesnastkowej ze znakiem ucieczki — \x1f), binarne 0 (w postaci ósemkowej ze znakiem ucieczki — \000) oraz d. Aby to sprawdzić, należy przekazać łańcuch znaków do funkcji `len` albo użyć funkcji `ord` do wyświetlenia kodów poszczególnych znaków. Więcej informacji na temat znaków ucieczki znajdziesz w tabeli 7.2.
7. Obecnie nie należy już używać modułu `string` zamiast metod łańcuchów znaków — sposób ten jest przestarzały, a wywołania tego typu całkowicie usunięto w Pythonie 3.x. Jedynym powodem uzasadniającym korzystanie z modułu `string` są jego inne narzędzia, takie jak zdefiniowane stałe. Odwołania do modułu `string` możesz także spotkać w bardzo starym i zakurzonym kodzie Pythona (i książkach napisanych w zamierzchłej przeszłości, gdzieś w latach 90. ubiegłego stulecia).

[1] Sekwencja ucieczki `\Uhhhh...` składa się z dokładnie ośmiu cyfr szesnastkowych (*h*); zarówno sekwencji `\u`, jak i `\U` w wersji 2.x można używać jedynie w literałach ciągów znaków Unicode, ale mogą być także używane w normalnych ciągach znaków w wersji 3.x (które z definicji są ciągami Unicode). W literałach typu `bytes` wersji 3.x szesnastkowe i ósemkowe sekwencje ucieczki reprezentują bajty o podanych wartościach; w literałach *łańcuchowych* takie sekwencje ucieczki oznaczają znak Unicode o podanym kodzie. Więcej szczegółowych informacji na temat sekwencji ucieczek Unicode znajdziesz w rozdziale 37.

[2] Osoby o większych zdolnościach matematycznych (a także studenci na moich kursach) czasami zauważają tutaj niewielką asymetrię. Element znajdujący się najbardziej na lewo ma wartość przesunięcia 0, natomiast element umieszczony najbardziej na prawo -1. Niestety, w Pythonie nie ma czegoś takiego jak odrębna wartość -0.

[3] Zobacz także uwagę z rozdziału 31. na temat błędów (lub raczej zmian) w metodzie `str.format` w Pythonach 3.2 i 3.3, dotyczących ogólnych pustych znaczników podstawienia dla atrybutów obiektów, które nie definiują obsługi funkcji `__format__`. Opisane zmiany miały negatywny wpływ na poprawnie wcześniej działający przykład z poprzedniej edycji tej książki. Chociaż mogą to być zmiany tymczasowe, to jasno podkreślają fakt, że ta metoda wciąż jest w fazie rozbudowy — jest to kolejny powód, aby kwestionować potrzebę jej wprowadzania i używania.

Rozdział 8. Listy oraz słowniki

Do tej pory omówiliśmy już liczby i ciągi znaków, a w tym rozdziale zaprezentujemy obiekty *list* oraz *słowników*, które są tak naprawdę kolekcjami innych obiektów. Te dwa typy są lokomotywami ciągnącymi niemal wszystkie skrypty napisane w Pythonie. Jak zobaczymy, oba typy są też zadziwiająco elastyczne — można je zmieniać w miejscu, rozszerzać i zmniejszać na żądanie; obydwa mogą również zawierać dowolne inne typy obiektów i mogą być osadzane w innych typach obiektów. Korzystając z tych typów, w skryptach można tworzyć i przetwarzanie dowolnie bogate struktury informacji.

Listy

Kolejnym przystankiem w naszym omówieniu wbudowanych typów obiektów Pythona są *listy*. Listy są najbardziej elastycznym typem obiektu uporządkowanej kolekcji. W przeciwieństwie do łańcuchów znaków listy mogą zawierać dowolne rodzaje obiektów — liczby, łańcuchy znaków, a nawet inne listy. W przeciwieństwie do łańcuchów znaków listy można również modyfikować w miejscu poprzez przypisanie do pozycji przesunięcia i wycinków, wywołania metod list czy instrukcje usuwające elementy. Są to obiekty *mutowalne*.

Listy w Pythonie wykonują większość pracy na strukturach danych kolekcji, które w językach niższego poziomu, jak na przykład C, implementować trzeba ręcznie. Poniżej znajduje się przegląd najważniejszych właściwości list.

Listy są uporządkowanymi kolekcjami dowolnych obiektów

Z funkcjonalnego punktu widzenia listy są miejscami zbierającymi inne obiekty, przez co można je traktować jak grupy. Zachowują one również uporządkowanie elementów od lewej do prawej strony (są zatem sekwencjami).

Dostęp do elementów list można uzyskać za pomocą pozycji przesunięcia

Tak jak w przypadku łańcuchów znaków, element listy można z niej pobrać, indeksując listę w pozycji przesunięcia obiektu. Ponieważ elementy listy są uporządkowane pozycyjnie, możliwe jest wykonywanie wycinków czy konkatenacja.

Listy mają zmienną długość, są niejednorodne i można je dowolnie zagnieżdżać

W przeciwieństwie do łańcuchów znaków listy mogą rosnąć i kurczyć się w miejscu, czyli ich długość może się zmieniać. Mogą również zawierać dowolny typ obiektów, nie tylko jednoznakowe łańcuchy — są zatem niejednorodne (heterogeniczne). Ponieważ listy mogą zawierać inne skomplikowane obiekty, obsługują również dowolne zagnieżdżanie. W Pythonie możliwe jest tworzenie list składających się z innych list.

Listy należą do sekwencji mutowalnych

Jeżeli chodzi o przydział do odpowiedniej kategorii, listy można modyfikować w miejscu (są zmienne) i reagują na wszystkie operacje na sekwencjach wykorzystywane z łańcuchami znaków, takie jak indeksowanie, wycinki i konkatenacja. Operacje na sekwencjach działają na listach w ten sam sposób jak na łańcuchach znaków. Jedyną różnicą jest to, że operacje takie, jak konkatenacja i wycinki po zastosowaniu do list zwracają nowe listy, a nie nowe łańcuchy znaków. Ponieważ listy są mutowalne, obsługują również inne operacje, których

nie obsługują łańcuchy znaków (na przykład operacje usuwania czy przypisania do indeksu, zmieniające listę w miejscu).

Listy są tablicami referencji do obiektów

Z technicznego punktu widzenia listy zawierają zero lub większą liczbę referencji do innych obiektów. Listy mogą przypominać nam tablice wskaźników (adresów). Pobranie elementu z listy Pythona jest prawie tak szybkie jak zindeksowanie tablicy języka C. Tak naprawdę wewnątrz standardowego interpretera Pythona listy są tablicami z języka C, a nie połączonymi strukturami. Jak jednak wspominaliśmy w rozdziale 6., Python zawsze śledzi referencję do obiektu za każdym jej użyciem, dlatego program ma do czynienia jedynie z obiektemi. Kiedy przypiszemy obiekt do komponentu struktury danych czy nazwy zmiennej, Python zawsze przechowuje referencję do samego obiektu, a nie jego kopię (o ile tego w jawnym sposobie nie zażdamy).

W tabeli 8.1 przedstawiono reprezentatywną listę często wykorzystywanych operacji na listach. Jest to niemal kompletna lista dla Pythona w wersji 3.3, ale jak zawsze więcej informacji możesz znaleźć w dokumentacji biblioteki standardowej Pythona lub wywołując w sesji interaktywnej funkcje `help(list)` bądź `dir(list)` w celu uzyskania pełnej listy metod list. Do funkcji tych można przekazać zarówno prawdziwą, istniejącą listę, jak i słowo kluczowe `list`, będące nazwą typu danych. Zestaw dostępnych metod jest szczególnie podatny na zmiany — na przykład w Pythonie 3.3 pojawiły się dwie nowe metody.

Tabela 8.1. Często stosowane literały list oraz operacje na tym typie danych

Operacja	Interpretacja
<code>L = []</code>	Pusta lista
<code>L = [123, 'abc', 1.23, {}]</code>	Cztery elementy — indeksy od 0 do 3
<code>L = ['Adam', 40.0, ['dev', 'mgr']]</code>	Zagnieżdżone podlisty
<code>L = list('mielonka')</code> <code>L = list(range(-4, 4))</code>	Lista elementów obiektu iterowanego Lista kolejnych liczb całkowitych
<code>L[i]</code> <code>L[i][j]</code> <code>L[i:j]</code> <code>len(L)</code>	Indeks, indeks indeksu, wycinek, długość
<code>L1 + L2</code> <code>L * 3</code>	Konkatenacja, powtórzenie
<code>for x in L: print(x)</code> <code>3 in L</code>	Iteracja, przynależność
<code>L.append(4)</code> <code>L.extend([5,6,7])</code> <code>L.insert(i,X)</code>	Metody: dodawanie elementów
<code>L.index(X)</code> <code>L.count(X)</code>	Metody: przeszukiwanie
<code>L.sort()</code>	Metody: sortowanie, odwracanie, kopiowanie (wersja 3.3+),

L.reverse() L.copy() L.clear()	czyszczenie (wersja 3.3+)
L.pop(i) L.remove(X) del L[i] del L[i:j] L[i:j] = []	Metody, wyrażenia: zmniejszanie listy
L[i] = 3 L[i:j] = [4,5,6]	Przypisanie do indeksu, przypisanie do wycinka
L = [x**2 for x in range(5)] List(map(ord, 'mielonka'))	Listy składane i odwzorowania (rozdziały 4., 14. oraz 20.)

Kiedy listę zapiszemy jako wyrażenie z literałem, zostaje ona zakodowana jako rozdzielona przecinkami seria obiektów (a tak naprawdę wyrażeń zwracających obiekty) w nawiasach kwadratowych, oddzielone przecinkami. Na przykład drugi wiersz z tabeli 8.1 przypisuje zmienną L do listy czteroelementowej. Osadzona lista kodowana jest jako seria elementów w nawiasach kwadratowych (rząd trzeci), natomiast pusta lista to po prostu para nawiasów kwadratowych niezawierająca niczego (pierwszy wiersz) [\[1\]](#).

Wiele z operacji przedstawionych w tabeli 8.1 powinno wyglądać znajomo, ponieważ są one tymi samymi działaniami, jakie widzieliśmy przy omawianiu łańcuchów znaków — jak indeksowanie, konkatenacja czy iteracje. Listy reagują również na wywołania metod specyficznych dla tego typu danych (które udostępniają narzędzia takie, jak sortowanie, odwracanie, dodawanie nowych elementów na końcu) oraz operacje modyfikujące te obiekty w miejscu (usuające elementy list, przypisujące coś do indeksów oraz wycinków). Listy otrzymują narzędzia obsługujące operacje modyfikacji, ponieważ są mutowalnym typem obiektów.

Listy w akcji

Chyba najlepszym sposobem na zrozumienie list jest zobaczenie ich w akcji. Zajmijmy się znowu prostymi kodami z sesji interaktywnej, które pomogą nam zilustrować operacje z tabeli 8.1.

Podstawowe operacje na listach

Listy reagują na operatory + oraz * podobnie jak łańcuchy znaków. Tutaj również oznaczają one konkatenację i powtórzenie, jednak rezultatem będzie nowa lista, a nie nowy łańcuch znaków. Listy obsługują wszystkie uniwersalne operacje na sekwencjach, jakie w poprzednim rozdziale wypróbowaliśmy na łańcuchach znaków.

```
% python
>>> len([1, 2, 3]) # Długość
3
```

```

>>> [1, 2, 3] + [4, 5, 6]                      # Konkatenacja
[1, 2, 3, 4, 5, 6]
>>> ['Ni!] * 4                                # Powtórzenie
['Ni!', 'Ni!', 'Ni!', 'Ni!']

```

Choć operator `+` działa tak samo dla list i łańcuchów znaków, należy pamiętać, że oczekuje on sekwencji tego samego typu po obu stronach — w przeciwnym razie po wykonaniu kodu otrzymamy błąd typu. Nie można zatem dokonać konkatenacji listy i łańcucha znaków bez wcześniejszej konwersji listy do łańcucha znaków (dzięki apostrofom lewym, `str` lub formatowaniu z `%`) albo łańcucha do listy (dzięki wbudowanej funkcji `list`).

```

>>> str([1, 2]) + "34"                         # To samo co "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list("34")                        # To samo co [1, 2] + ["3", "4"]
[1, 2, '3', '4']

```

Iteracje po listach i składanie list

Listy mogą być używane we wszystkich operacjach na sekwencjach, jakie poznaliśmy w poprzednim rozdziale przy okazji omawiania ciągów znaków, dotyczy to również narzędzi iteracyjnych:

```

>>> 3 in [1, 2, 3]                            # Przynależność
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')
print x,)                                     # Iteracja (w wersji 2.x używamy
...
1 2 3

```

Więcej informacji na temat iteracji w pętli `for` oraz funkcji wbudowanej `range` pokażemy w rozdziale 13., ponieważ są to zagadnienia związane ze składnią instrukcji. W skrócie: pętla `for` pozwala na przetwarzanie elementów sekwencji po kolej, wykonując w każdej iteracji jedną lub większą liczbę instrukcji; funkcja `range` zwraca listy zawierające kolejne liczby całkowite.

Ostatnie elementy tabeli 8.1, czyli listy składane i wywołania funkcji `map`, zostaną wstępnie omówione w rozdziale 14. oraz bardziej szczegółowo w rozdziale 20. Ich działanie jest jednak dość proste: jak wspomniałem w rozdziale 4., listy składane są sposobem budowania nowej listy przez wywołanie wyrażenia na elementach sekwencji (tak jak to robimy w przypadku innych obiektów iterowalnych). W filozofii działania listy składane przypominają pętle `for`:

```

>>> res = [c * 4 for c in 'JAJKO']           # Lista składana
>>> res
['JJJJ', 'AAAA', 'JJJJ', 'KKKK', '0000']

```

To wyrażenie jest właściwie równoważne pętli `for` składającej listę wyników, ale jak będziemy mieli okazję przekonać się w dalszych rozdziałach, listy składane tworzy się łatwiej i z reguły działają one szybciej.

```
>>> res = []
```

```

>>> for c in 'JAJKO':
...     res.append(c * 4)
...
>>> res
['JJJJ', 'AAAA', 'JJJJ', 'KKKK', '0000']

```

Jak również wspominałem w rozdziale 4., funkcja wbudowana `map` ma podobne działanie, ale zamiast wyrażeń na elementach sekwencji wywoływane są funkcje, a wyniki zwracane są w postaci nowej listy:

```

>>> list(map(abs, [-1, -2, 0, 1, 2]))          # Wywołanie funkcji map na
                                                sekwencji
[1, 2, 0, 1, 2]

```

Na tym etapie książki nie jesteśmy jeszcze gotowi na przekazanie pełnych informacji dotyczących iteratorów, zatem odłożymy to zagadnienie do późniejszych rozdziałów. W dalszej części niniejszego rozdziału wróćmy jednak na chwilę do tego zagadnienia przy okazji podobnych wyrażeń słowników składanych.

Indeksowanie, wycinki i macierze

Ponieważ listy są sekwencjami, indeksowanie i wycinki działają w ten sam sposób dla list, jak i dla łańcuchów znaków. Wynikiem zindeksowania listy jest jednak dowolny typ obiektu znajdujący się na pozycji o podanej wartości przesunięcia, natomiast wycinek listy zawsze zwraca nową listę.

```

>>> L = ['mielonka', 'Mielonka', 'MIELONKA!']
>>> L[2]                                         # Wartości przesunięcia
                                                rozpoczętają się od 0
'MIELONKA!'
>>> L[-2]                                        # Wartość ujemna: odliczamy od
                                                końca
'Mielonka'
>>> L[1:]                                         # Wycinek pobiera części listy
['Mielonka', 'MIELONKA!']

```

Jedna uwaga: ponieważ wewnątrz list można zagnieździć inne listy (oraz inne typy), czasami w celu wejścia w głęb struktury danych niezbędne będzie połączenie ze sobą kilku operacji indeksowania. Jednym z łatwiejszych sposobów reprezentowania w Pythonie macierzy (tablic wielowymiarowych) są listy z zagnieżdżonymi podlistami. Poniżej widać prostą dwuwymiarową tablicę 3×3 opartą na listach.

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Jeden indeks pozwala nam pobrać cały wiersz (a tak naprawdę zagnieżdżoną sublistę), natomiast dwa indeksy — pojedynczy element tego wiersza.

```

>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]

```

5

```
>>> matrix[2][0]
7
>>> matrix = [[1, 2, 3],
...             [4, 5, 6],
...             [7, 8, 9]]
>>> matrix[1][1]
5
```

W powyższym kodzie widać, że listy w naturalny sposób mogą się rozciągać na kilka wierszy, jeśli tego chcemy, ponieważ są ujęte w parę nawiasów kwadratowych; wielokropek ... reprezentuje znaki zachęty wierszy kontynuacji kodu (porównywalny kod bez znaków zachęty ... znajdziesz w rozdziale 4.; więcej szczegółowych informacji na temat składni znajdziesz w następnej części książki).

Więcej informacji na temat macierzy znajdziesz w dalszej części tego rozdziału, gdzie będziemy omawiać słownikową reprezentację macierzy, która może być bardziej wydajna w przypadku, gdy macierze są w dużej mierze puste. Będziemy również kontynuować ten wątek w rozdziale 20., w którym napiszemy dodatkowy kod przetwarzający macierze, ze szczególnym uwzględnieniem list składanych. W przypadku zadań wymagających dużych ilości obliczeń numerycznych możesz skorzystać z rozszerzenia *NumPy*, zapewniającego inne sposoby obsługi macierzy; wspominaliśmy o nim w rozdziale 4. i rozdziale 5.

Modyfikacja list w miejscu

Ponieważ listy są typem mutowalnym, obsługują operacje zmieniające obiekt listy *w miejscu*. Oznacza to, że wszystkie operacje wymienione poniżej modyfikują bezpośrednio obiekt listy, nadpisując jego poprzednią wartość i nie zmuszając nas do tworzenia nowej kopii, tak jak było to w przypadku łańcuchów znaków. Ponieważ w Pythonie mamy do czynienia z referencjami do obiektów, rozróżnienie między zmianą obiektu w miejscu a utworzeniem nowego obiektu ma znaczenie. Jak wspomiano w rozdziale 6., jeżeli zmienimy obiekt w miejscu, może to mieć wpływ na większą liczbę referencji do niego.

Przypisywanie do indeksu i wycinków

Kiedy używamy list, możemy zmieniać ich zawartość, przypisując nowe wartości do określonego elementu (na pozycji o podanym przesunięciu) lub całego fragmentu (wycinka).

```
>>> L = ['mielonka', 'Mielonka', 'MIELONKA!']
>>> L[1] = 'jajka'                                     # Przypisanie do indeksu
>>> L
['mielonka', 'jajka', 'MIELONKA!']
>>> L[0:2] = ['najsmaczniejsza', 'jest']           # Przypisanie do wycinka:
usunięcie i wstawienie
>>> L
['najsmaczniejsza', 'jest', 'MIELONKA!']
```

Przypisanie do indeksu i wycinka to zmiana w miejscu. Operacje te modyfikują listę w sposób bezpośredni, a nie generują nowy obiekt listy dla wyniku. Przypisanie do indeksu w Pythonie

działa mniej więcej tak, jak w C i innych językach programowania — Python zastępuje odwołanie do pojedynczego obiektu na określonej pozycji przesunięcia nową referencją.

Przypisanie do wycinka, czyli ostatnia operacja z powyższego przykładu, zastępuje wybraną część listy za jednym razem. Ponieważ może być nieco skomplikowana, zatem najlepiej ją sobie wyobrazić jako połączenie dwóch kroków:

1. *Usunięcie*. Wycinek podany po lewej stronie znaku = jest usuwany.
2. *Wstawienie*. Nowe elementy znajdujące się w iterowalnym obiekcie po prawej stronie znaku = wstawiane są do listy po lewej stronie, w miejscu, z którego wcześniej usunęliśmy stary wycinek^[2].

W rzeczywistości nie wygląda to dokładnie tak, ale taki opis pozwala wyjaśnić, dlaczego liczba wstawianych elementów nie musi odpowiadać liczbie elementów usuwanych. Na przykład, kiedy mamy listę L, składającą się z dwóch lub więcej elementów, przypisanie L[1:2] = [4,5] zastępuje jeden element dwoma — Python najpierw usuwa wycinek jednoelementowy [1:2], a później w miejsce usuniętego wycinka wstawia 4 i 5.

Wyjaśnia to również, dlaczego drugie przypisanie wycinka poniżej jest tak naprawdę wstawianiem — Python zastępuje pusty wycinek w [1: 1] dwoma elementami; i dlaczego trzecie jest tak naprawdę usunięciem — Python usuwa wycinek (element z przesunięciem 1), a następnie nie wstawia nic na jego miejsce:

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5]                                     # Zamiana / wstawianie
>>> L
[1, 4, 5, 3]
>>> L[1:1] = [6, 7]                                     # Wstawianie (nic nie zamieniamy)
>>> L
[1, 6, 7, 4, 5, 3]
>>> L[1:2] = []                                       # Usuwanie (nic nie wstawiamy)
>>> L
[1, 7, 4, 5, 3]
```

W rezultacie przypisanie do wycinka zastępuje cały fragment, czy inaczej „kolumnę”, za jednym razem. Ponieważ długość przypisywanej sekwencji nie musi odpowiadać długości wycinka, do którego ją przypisujemy, operację tę można wykorzystać do zastąpienia (poprzez nadpisanie), rozszerzenia (poprzez wstawienie) i zmniejszenia (poprzez usunięcie) listy. Ta technika ma duże możliwości, jednak szczerze mówiąc, nieczęsto spotyka się ją w praktyce. Zazwyczaj istnieją łatwiejsze sposoby zastępowania, wstawiania i usuwania (na przykład konkatenacja i metody list insert, pop oraz remove), które wolą stosować programiści Pythona.

Z drugiej strony taka operacja może być użyta jako rodzaj konkatenacji w miejscu na początku listy — analogicznie do tego, co robi metoda extend na końcu listy, o czym opowiemy bardziej szczegółowo już w kolejnym podrozdziale:

```
>>> L = [1]
>>> L[:0] = [2, 3, 4]                                 # Wstaw wszystko na pozycji :0, pusty
wycinek na początku
>>> L
```

```

[2, 3, 4, 1]
>>> L[len(L):] = [5, 6, 7]          # Wstaw wszystko na pozycji len(L):,
pusty wycinek na końcu

>>> L
[2, 3, 4, 1, 5, 6, 7]
>>> L.extend([8, 9, 10])           # Wstaw wszystko na końcu; używamy
metody extend

>>> L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]

```

Wywołania metod list

Podobnie jak w przypadku łańcuchów znaków, obiekty list w Pythonie obsługują wywołania metod specyficznych dla tego typu obiektu, z których wiele może zmieniać listę w miejscu.

```

>>> L = ['najsmaczniejsza', 'jest', 'MIELONKA!']

>>> L.append('puszkowana')          # Dodanie elementu na końcu
listy

>>> L
['najsmaczniejsza', 'jest', 'MIELONKA!', 'puszkowana']

>>> L.sort()                      # Sortowanie listy ('M' < 'j')

>>> L
['MIELONKA!', 'jest', 'najsmaczniejsza', 'puszkowana']

```

Metody zostały przedstawione w rozdziale 7. Mówiąc w skrócie, są one funkcjami (tak naprawdę — atrybutami odnoszącymi się do funkcji), które powiązane są z określonymi obiektami. Metody udostępniają narzędzia specyficzne dla danego typu. Zaprezentowane tutaj metody list są na przykład dostępne wyłącznie dla list.

Chyba najczęściej wykorzystywana metodą list jest `append`, wstawiająca pojedynczy element (referencję do obiektu) na koniec listy. W przeciwnieństwie do konkatenacji `append` oczekuje przekazania pojedynczego obiektu, a nie listy. Rezultat działania `L.append(X)` jest podobny do `L+[X]`, jednak podczas gdy pierwszy ze sposobów modyfikuje listę `L` w miejscu, drugi tworzy nowy obiekt listy [\[3\]](#). Metoda `sort` porządkuje tutaj kolejność elementów listy, ale z pewnością zasługuje na własny podrozdział.

Kilka słów o sortowaniu list

Inna popularna metoda, `sort`, porządkuje elementy listy w miejscu. Domyślnie wykorzystuje standardowe testy porównania Pythona (mamy tutaj porównywanie łańcuchów znaków, ale dotyczy to obiektów każdego typu) i domyślnie sortuje listę w kolejności rosnącej. Działanie funkcji `sort` można modyfikować, wykorzystując *argumenty ze słowami kluczowymi* (z użyciem specjalnej składni w postaci par *nazwa=wartość*, pozwalającej na przekazywanie wybranych argumentów funkcji, która jest często stosowana w Pythonie do przekazywania opcji konfiguracyjnych).

W wywołaniach metody `sort` argument `reverse` pozwala na sortowanie w kolejności malejącej zamiast rosnącej, a argument `key` tworzy funkcję jednoargumentową zwracającą wartość, która zostanie użyta podczas sortowania — w poniższym przykładzie argument ten został użyty do wywołania metody `lower`, spełniającej rolę konwertera znaków łańcucha na małe litery (warto

pamiętać, że jej nowszy odpowiednik, metoda `casifold`, znacznie lepiej obsługuje niektóre typy tekstu Unicode):

```
>>> L = ['abc', 'ABD', 'aBe']  
>>> L.sort()                                     # Sortowanie bez uwzględnienia  
wielkości liter  
>>> L  
['ABD', 'aBe', 'abc']  
>>> L = ['abc', 'ABD', 'aBe']  
>>> L.sort(key=str.lower)                         # Normalizacja do małych liter  
>>> L  
['abc', 'ABD', 'aBe']  
>>>  
>>> L = ['abc', 'ABD', 'aBe']  
>>> L.sort(key=str.lower, reverse=True)           # Zmiana kolejności sortowania  
>>> L  
['aBe', 'ABD', 'abc']
```

Argument `key` metody `sort` może być również przydatny podczas sortowania list słowników, do wybierania klucza sortowania poprzez indeksowanie każdego słownika. Słowniki przestudujemy w dalszej części tego rozdziału, a więcej o argumentach funkcji, zawierających słowa kluczowe, dowiesz się w części IV tej książki.



Porównywanie i sortowanie w wersji 3.x: w Pythonie 2.x bez problemu działa porównywanie względnych wielkości obiektów różnych typów (np. ciągu znaków i listy), jak to pokazywaliśmy w rozdziale 5. — język ma zdefiniowany algorytm określania porządku danych różnych typów, który jest deterministyczny, choć być może mało elegancki. Oznacza to, że kolejność jest zdefiniowana w oparciu o kolejność alfabetyczną nazw typów, czyli liczby są mniejsze od ciągów znaków, ponieważ `int` jest w alfabetie wcześniej niż `str`. Porównanie nigdy nie wykonuje konwersji typów, z wyjątkiem porównywania obiektów typu liczbowego.

W Pythonie 3.x to uległo zmianie: próba porównania wielkości różnych typów zgłasza wyjątek, zamiast dawać wyniki oparte na stałej liście kolejności typów. Ponieważ operacja sortowania wewnętrznie wykorzystuje operację porównywania, oznacza to, że wyrażenie `[1, 2, 'spam'].sort()` zadziała w Pythonie 2.x, ale w Pythonie 3.x i nowszych spowoduje zgłoszenie wyjątku.

W Pythonie 3.x zrezygnowano z możliwości przekazywania do funkcji `sort` dowolnej funkcji porównującej elementy. Sugerowanym obejściem tego braku jest użycie argumentu ze słowem kluczowym `key=func`, służącego do przekształcania wartości przed sortowaniem, i używanie argumentu `reverse=True` w celu odwrócenia kolejności sortowania. Takie rozwiązanie było również powszechnie stosowane w przeszłości.

Jedno ostrzeżenie: pamiętaj, że metody `append` oraz `sort` modyfikują powiązany obiekt listy w miejscu, jednak nie zwracają listy jako wyniku (z technicznego punktu widzenia zwracając wartość o nazwie `None`). Jeżeli napiszemy coś podobnego do `L = L.append(X)`, nie otrzymamy zmodyfikowanej wartości `L` (a tak naprawdę całkowicie stracimy referencję do listy). Kiedy

używasz atrybutów takich jak `append` czy `sort`, efektem ubocznym jest modyfikacja obiektów, dlatego nie ma potrzeby ponownego ich przypisywania.

Częściowo z powodu tego typu ograniczeń operacja sortowania jest w nowszych wersjach Pythona dostępna również w postaci funkcji wbudowanej `sorted`, która może być użyta do sortowania dowolnej sekwencji (nie tylko list) i zwraca nową listę (zamiast modyfikowania obiektu źródłowego).

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L, key=str.lower, reverse=True)          # Wbudowana funkcja
sortująca
['aBe', 'ABD', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True)    # Wstępne przekształcenie
elementów: zmienione
                                                # elementy wyniku!
['abe', 'abd', 'abc']
```

Warto zwrócić uwagę na ostatni przykład: dane wejściowe przekształcamy przed sortowaniem, wykorzystując wyrażenie listy składanej, co powoduje, że wynik nie zawiera oryginalnych wartości, jak to ma miejsce w przypadku użycia argumentu `key`. W przypadku tego ostatniego dane są przekształcane tymczasowo na potrzeby porównań, ale w wyniku zwracana jest ich oryginalna postać. W dalszej części rozdziału przedstawię kilka zastosowań, w których funkcja wbudowana `sorted` sprawdzi się lepiej od metody `sort`.

Inne, często stosowane metody list

Podobnie jak łańcuchy znaków, listy mają również inne metody wykonujące różne wyspecjalizowane operacje. Na przykład metoda `reverse` odwraca listę w miejscu, metoda `extend` wstawia kilka elementów na końcu listy, a `pop` usuwa element znajdujący się na końcu listy.

Istnieje również wbudowana funkcja `reversed`, która działa podobnie do funkcji `sorted` i zwraca nowy obiekt wynikowy, ale musi być opakowana w wywołanie metody `list` zarówno w wersji 2.x, jak i 3.x, ponieważ jej wynikiem jest iterator, który generuje wyniki na żądanie (więcej informacji o iteratorach znajdziesz w dalszej części książki):

```
>>> L = [1, 2]
>>> L.extend([3,4,5])          # Dodanie kilku elementów na
końcu (jak rozszerzenie w miejscu)
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                  # Usunięcie i zwrócenie
ostatniego elementu (domyślnie -1)
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()              # Odwrócenie kolejności w
miejscu
```

```

>>> L
[4, 3, 2, 1]
>>> list(reversed(L))          # Wbudowana funkcja odwracająca
sekwencję (zwracająca iterator)
[1, 2, 3, 4]

```

Technicznie rzecz biorąc, metoda `extend` zawsze dokonuje iteracji i dodaje każdy element w obiekcie *iterowalnym*, podczas gdy metoda `append` dodaje po prostu pojedynczy element, tak jak jest, bez iterowania — rozróżnienie, które będzie bardziej znaczące w rozdziale 14. Na razie powinieneś po prostu zapamiętać, że metoda `extend` dodaje wiele elementów, a `append` dodaje tylko jeden. W pewnych rodzajach programów wykorzystana powyżej metoda `pop` jest często używana w połączeniu z `append` do szybkiego zaimplementowania struktury *stosu* typu LIFO (ang. *last-in-first-out*). Koniec listy służy w takiej sytuacji za wierzchołek stosu.

```

>>> L = []
>>> L.append(1)                # Wstawienie na stos
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                   # Usunięcie ze stosu
2
>>> L
[1]

```

Metoda `pop` akceptuje także opcjonalny argument `offset`, służący do określania indeksu elementu, który ma być usunięty i zwrócony (domyślnie jest to ostatni element, znajdujący się na przesunięciu `-1`). Istnieją również metody usuwające elementy według ich wartości (`remove`), wstawiające element we wskazanym miejscu (`insert`), zliczające liczbę wystąpień (`count`) i wyszukujące pozycję elementu o zadanej wartości (metoda `index` wyszukuje indeks elementu, nie należy mylić takiej operacji z indeksowaniem!):

```

>>> L = ['mielonka', 'jajka', 'szynka']
>>> L.index('jajka')          # Pozycja elementu o zadanej wartości
(wyszukiwanie)
1
>>> L.insert(1, 'tost')       # Wstawianie na określonej pozycji
>>> L
['mielonka', 'tost', 'jajka', 'szynka']
>>> L.remove('jajka')        # Usunięcie elementu o zadanej wartości
>>> L
['mielonka', 'tost', 'szynka']
>>> L.pop(1)                  # Usunięcie elementu na zadanej pozycji
'tost'
>>> L

```

```
['mielonka', 'szynka']
>>> L.count('mielonka')                                # Zliczanie liczby wystąpień
1
```

Zwróć uwagę, że w przeciwieństwie do innych metod listy, metody `count` i `index` nie zmieniają samej listy, ale zwracają informacje o jej zawartości. Więcej informacji na temat metod obiektów list możesz znaleźć w dokumentacji Pythona, warto też poeksperymentować z nimi w konsoli interaktywnej.

Inne popularne operacje na listach

Ponieważ listy są typami mutowalnymi, możesz użyć instrukcji `del` do usunięcia elementu lub części listy w miejscu.

```
>>> L
['MIELONKA!', 'jest', 'najsmaczniejsza', 'puszkowana']
>>> del L[0]                                         # Usunięcie jednego elementu
>>> L
['jest', 'najsmaczniejsza', 'puszkowana']
>>> del L[1:]                                         # Usunięcie całej części
>>> L
# To samo co L[1:] = []
['jest']
```

Jak już wspominaliśmy wcześniej, ponieważ przypisanie wycinka składa się z usunięcia i wstawienia, można również usunąć fragment listy, przypisując do wycinka pustą listę (`L[i:j] = []`). Python usuwa wycinek podany po lewej stronie, a następnie nie wstawia nic w jego miejsce. Z drugiej strony, przypisanie pustej listy do indeksu po prostu przechowuje referencję do pustego obiektu listy w określonym miejscu, zamiast coś usuwać.

```
>>> L = ['Mam', 'już', 'coś']
>>> L[1:] = []
>>> L
['Mam']
>>> L[0] = []
>>> L
[]
```

Chociaż wszystkie omówione powyżej operacje są typowe, istnieją również dodatkowe metody listy i operacje, których tutaj nie pokazano. Na przykład zestaw dostępnych metod może się zmieniać w czasie i tak się na przykład zdarzyło w Pythonie 3.3 — jego nowa metoda `L.copy()` tworzy kopię listy najwyższego poziomu, podobnie jak `L[:]` i `list(L)`, ale jest analogiczna do metody `copy` w zbiorach i słownikach. Wyczerpującą i zawsze aktualną listę narzędzi dla danego typu można znaleźć w dokumentacji Pythona za pomocą funkcji `dir` oraz `help` (z którymi spotkaliśmy się już w rozdziale 4.), a także w książce *Python. Leksykon kieszonkowy* (wydawnictwo Helion) i innych teksthach przedstawionych w przedmowie do niniejszej książki.

Chciałbym również przypomnieć raz jeszcze, że wszystkie omówione wyżej operacje modyfikujące obiekty w miejscu działają jedynie na obiektach mutowalnych i nie będą działać na łańcuchach znaków (ani omówionych w rozdziale 9. krotkach) — bez względu na to, jak

bardzo będziemy się o to starać. Mutowalność jest nieodłączną właściwością każdego typu obiektu.

Słowniki

Obok list, *słowniki* (ang. *dictionary*) są jednym z najbardziej elastycznych wbudowanych typów danych w Pythonie. Jeżeli wyobrażymy sobie listy jako uporządkowane kolekcje obiektów, słowniki będą kolekcjami nieuporządkowanymi. Podstawowa różnica między tymi dwoma typami danych polega na tym, że w słownikach elementy są przechowywane i pobierane według *kluczy*, a nie pozycji przesunięcia. Podczas gdy listy mogą pełnić rolę podobną do tablic w innych językach programowania, słowniki zastępują rekordy, tabele wyszukiwania i wszelkie inne rodzaje agregacji, w których nazwy przedmiotów są bardziej znaczące niż ich pozycje.

Na przykład słowniki mogą zastąpić wiele mechanizmów wyszukiwania i struktur danych, które w językach niższego poziomu musielibyśmy implementować ręcznie — ponieważ słowniki są wysoce zoptymalizowanym typem wbudowanym, ich indeksowanie jest bardzo szybką operacją wyszukiwania. Słowniki czasami spełniają również rolę rekordów i tablic symboli z innych języków programowania. Mogą też na przykład reprezentować tzw. rzadkie struktury danych (ang. *sparse data structures*), czyli takie, w których większość elementów jest pusta. Poniżej zamieszczamy krótkie podsumowanie ich właściwości.

Dostęp do słowników odbywa się według klucza, a nie wartości przesunięcia

Słowniki czasami nazywane są *tablicami asocjacyjnymi* lub *tablicami mieszającymi* (ang. *associative array* lub *hash*), zwłaszcza przez użytkowników korzystających z innych języków programowania. Wiążą one zbiór wartości z kluczami, tak by element słownika można było pobrać za pomocą klucza, pod którym jest on przechowywany. Do pobrania komponentów słownika wykorzystuje się tę samą operację indeksowania co w przypadku list, jednak indeks ma tutaj postać klucza, a nie względnej wartości przesunięcia.

Słowniki są nieuporządkowanymi kolekcjami dowolnych obiektów

W przeciwnieństwie do listy, elementy przechowywane w słowniku nie są przechowywane w żadnej szczególnej kolejności. Tak naprawdę kolejność elementów słownika jest w Pythonie pseudolosowa, dzięki czemu można je szybciej przeszukiwać. Klucze udostępniają symboliczną (a nie fizyczną) lokalizację elementów w słowniku.

Słowniki mają zmienną długość, są heterogeniczne i mogą być dowolnie zagnieżdżane

Tak jak listy, słowniki mogą rozszerzać się i kurczyć w miejscu (bez tworzenia nowych kopii) i mogą zawierać obiekty dowolnego typu. Obsługują również zagnieżdżanie na dowolną głębokość (mogą zawierać listy, inne słowniki itp.). Każdy *klucz* może mieć tylko jedną powiązaną wartość, ale w razie potrzeby taka wartość może być *zbiorem* wielu obiektów, każdą wartość można zapisać pod dowolną liczbą kluczy.

Słowniki należą do kategorii mutowalnych odwzorowań

Słowniki można zmieniać w miejscu przypisując je do indeksów (które są mutowalne). Nie obsługują jednak operacji sekwencyjnych, działających na łańcuchach znaków oraz listach. Ponieważ słowniki są kolekcjami nieuporządkowanymi, wszystkie operacje opierające się na stałej kolejności elementów (na przykład konkatenacja czy wycinki) nie mają w ich przypadku większego sensu. Zamiast tego słowniki są jedynym wbudowanym typem, będącym przedstawicielem kategorii *odwzorowań* (obiekty odwzorowujące klucze na wartości). Inne odwzorowania są w Pythonie tworzone poprzez importowane moduły.

Słowniki są tabelami referencji do obiektów (tablicami asocjacyjnymi)

Jeżeli listy są tablicami referencji do obiektów obsługującymi dostęp za pomocą pozycji elementów, słowniki to nieuporządkowane tabele referencji do obiektów obsługujące dostęp za pomocą kluczy. Wewnętrznie słowniki zaimplementowane są jako tablice asocjacyjne (struktury danych obsługujące bardzo szybkie pobieranie), które mogą rozszerzać się i kurczyć na żądanie. Co więcej, Python wykorzystuje zoptymalizowane algorytmy haszujące, służące do odnajdywania kluczy, dzięki czemu to pobieranie jest naprawdę szybkie. Podobnie jak listy, tak i słowniki przechowują referencje do obiektów (a nie ich kopie, o ile nie zdefiniujesz tego w sposób jawnym).

W tabeli 8.2 przedstawione zostały najczęściej wykorzystywane i najbardziej reprezentatywne operacje na słownikach, a ich lista jest niemal kompletna dla wersji 3.3 Pythona. Jak zawsze pełną listę można znaleźć w dokumentacji Pythona, a także wywołując funkcje `dir(dict)` lub `help(dict)`, gdzie `dict` to nazwa typu słownika. Kiedy słownik zakodujemy w postaci wyrażenia literałowego, jest zapisywany w postaci serii par `klucz:wartość`, rozdzielonych przecinkami i umieszczonych w nawiasach klamrowych^[4]. Pusty słownik jest zapisywany w postaci pustej pary nawiasów klamrowych. Słowniki można zagnieżdżać, wstawiając je jako wartość wewnętrznie innego słownika, listy lub krotki.

Tabela 8.2. Popularne literały i operacje słowników

Operacja	Interpretacja
<code>D = {}</code>	Pusty słownik
<code>D = { 'name': 'Adam', 'age': 40}</code>	Słownik dwuelementowy
<code>E = { 'cto': { 'name': 'Adam', 'age': 40}}</code>	Zagnieżdżanie
<code>D = dict(name='Adam', age=40)</code>	Alternatywne techniki tworzenia:
<code>D = dict([('name', 'Adam'), ('age', 40)])</code> <code>D = dict(zip(keyslist, valueslist))</code> <code>D = dict.fromkeys(['name', 'age'])</code>	słowa kluczowe, pary klucz-wartość, spięte pary klucz-wartość, listy kluczy
<code>D['name']</code> <code>D['cto']['age']</code>	Indeksowanie po kluczu
<code>'age' in D</code>	Metody: sprawdzanie przynależności klucza
<code>D.keys()</code>	Metody: wszystkie klucze
<code>D.values()</code>	wszystkie wartości,
<code>D.items()</code>	wszystkie krotki klucz+wartość
<code>D.copy()</code>	kopiowanie
<code>D.clear()</code>	czyszczenie (usuwa wszystkie elementy)
<code>D.update(D2)</code>	łączenie według kluczy
<code>D.get(key, default?)</code>	pobieranie według klucza, wartości domyślne (lub

	None), gdy brak klucza
D.pop(key, default?)	usuwanie według klucza, wartości domyślne (lub błęd), gdy brak klucza
D.setdefault(key, default?)	pobieranie według klucza, wartości domyślne (lub None), gdy brak klucza
D.popitem()	usuwanie lub zwracanie pary (klucz, wartość) itp.
len(D)	Długość (liczba przechowywanych wpisów)
D[key] = 42	Dodawanie lub modyfikacja kluczy
del D[key]	Usuwanie elementu według klucza
list(D.keys()) D1.keys() & D2.keys()	Widoki słowników (Python 3.x)
D.viewkeys(), D.viewvalues()	Widoki słowników (Python 2.7)
D = {x: x*2 for x in range(10)}	Słowniki składane (Python 3.x, 2.7)

Słowniki w akcji

Jak sugeruje tabela 8.2, słowniki są indeksowane według klucza, a do zagnieżdżonych pozycji słownika można się odwoływać za pomocą serii indeksów (kluczy w nawiasach kwadratowych). Kiedy Python tworzy słownik, przechowuje elementy w dowolnej kolejności od lewej do prawej strony. Aby pobrać wartość ze słownika, należy podać powiązany z nią klucz, a nie jej pozycję względową. Wróćmy zatem do sesji interpretera Pythona, aby nieco bliżej zapoznać się z niektórymi operacjami na słownikach, zaprezentowanymi w tabeli 8.2.

Podstawowe operacje na słownikach

Zazwyczaj tworzymy słowniki za pomocą literałów oraz przechowujemy elementy i uzyskujemy do nich dostęp za pomocą kluczy.

```
% python
>>> D = {'mielonka': 2, 'szynka': 1, 'jajka': 3} # Utworzenie słownika
>>> D['mielonka']                                # Pobranie wartości po
kluczu
2
>>> D                                         # Kolejność zostaje
pomieszana
{'jajka': 3, 'mielonka': 2, 'szynka': 1}
```

W powyższym kodzie słownik przypisywany jest do zmiennej D. Wartością klucza 'mielonka' jest liczba całkowita 2 itd. Do zindeksowania słownika po kluczu wykorzystujemy tę samą składnię z nawiasami kwadratowymi, jaką była używana przy indeksowaniu list za pomocą przesunięcia, z tym że tutaj oznacza ona jednak dostęp według klucza, a nie pozycji elementu.

Warto również zwrócić uwagę na końcówkę tego przykładu — podobnie jak w przypadku zbiorów, kolejność elementów słownika *od lewej do prawej* strony prawie zawsze jest inna od ich początkowej kolejności. Jest to celowe — w celu zaimplementowania szybkiego wyszukiwania kluczy (inaczej mieszania lub haszowania, ang. *hashing*) muszą one zostać w pamięci ułożone w sposób losowy. Z tego powodu operacje zakładające stały porządek elementów od lewej do prawej strony (na przykład wycinki czy konkatenacja) nie mają zastosowania do słowników. Wartości można pobrać jedynie za pomocą klucza, a nie pozycji. Z technicznego punktu widzenia kolejność elementów słownika jest *pseudolosowa* — nie ma tutaj prawdziwej losowości (teoretycznie można ją rozszyfrować, analizując kod źródłowy Pythona i mając dużo czasu do zabicia), ale mimo to jest dość arbitralna i może różnić się w zależności od wersji Pythona i platformy, na której działa, a w Pythonie 3.3 może być różna nawet w kolejnych interaktywnych sesjach interpretera.

Wbudowana funkcja `len` działa również na słownikach. Zwraca ona liczbę elementów przechowywanych w słowniku lub — równoważnie — długość jego listy kluczy. Operator przynależności `in` pozwala sprawdzać istnienie klucza, natomiast metoda `keys` zwraca wszystkie klucze słownika zebrane w listę. Może ona być przydatna do sekwencyjnego przetwarzania słowników, jednak nie powinieneś przykładać zbyt dużej wagi do kolejności kluczy na takiej liście. Ponieważ wyniki działania metody `keys` mogą być traktowane jak normalna lista, zawsze możesz je posortować, jeżeli kolejność ma dla Ciebie jakieś znaczenie.

```
>>> len(D)                                # Liczba wpisów w słowniku
3
>>> 'szynka' in D                         # Alternatywne sprawdzanie
istnienia klucza
True
>>> list(D.keys())                        # Utworzenie nowej listy kluczy
słownika D
['jajka', 'mielonka', 'szynka']
```

Warto zwrócić uwagę na drugie wyrażenie w tym przykładzie. Jak wspominaliśmy wcześniej, test przynależności `in` wykorzystywany w łańcuchach znaków oraz listach będzie również działał na słownikach. Sprawdza on, czy klucz jest przechowywany w słowniku. Z technicznego punktu widzenia rozwiążanie to działa, ponieważ słowniki definiują *iteratory* kluczy i używają szybkiego wyszukiwania wszędzie tam, gdzie jest to możliwe. Inne typy udostępniają iteratory odzwierciedlające ich częste zastosowania. W przypadku plików istnieją na przykład iteratory ładujące ich zawartość wiersz po wierszu. Iteratory zostaną omówione bardziej szczegółowo w rozdziałach 14. oraz 20.

Warto zwrócić uwagę na ostatnie wyrażenie w tym przykładzie. W Pythonie 3.x wyniki działania metody `keys` musimy przekształcić na listę — w 3.x metoda `keys` zwraca obiekt *iterowalny*, a nie gotową listę. Wywołanie funkcji `list` powoduje wygenerowanie wszystkich elementów i przekształcenie ich na listę, dzięki czemu możemy je wyświetlić w sesji interaktywnej, choć takie wywołanie nie jest wymagane w innych kontekstach. W wersji 2.x metoda `keys` tworzy i zwraca rzeczywistą listę, zatem wywołanie funkcji `list` nie jest konieczne do wyświetlenia wyników. Więcej szczegółowych informacji na ten temat znajdziesz w dalszej części rozdziału.

Modyfikacja słowników w miejscu

Kontynuujmy naszą sesję interaktywną. Słowniki, podobnie jak listy, są mutowalne, zatem można je modyfikować, rozszerzać i kurczyć w miejscu bez tworzenia nowych słowników — wystarczy przypisać wartość do klucza lub utworzyć nowy wpis. Działa tutaj również instrukcja `del`, która usuwa wpis powiązany z kluczem podanym w indeksie. Warto także zwrócić uwagę

na zagnieźdżenie listy wewnątrz słownika (wartość klucza 'szynka'). Wszystkie typy kolekcji z Pythona mogą być wewnątrz siebie dowolnie zagnieźdzane.

```
>>> D
{'jajka': 3, 'mielonka': 2, 'szynka': 1}
>>> D['szynka'] = ['grillowanie', 'pieczenie', 'smażenie']      # Zmiana wpisu
(wartość = lista)
>>> D
{'jajka': 3, 'mielonka': 2, 'szynka': ['grillowanie', 'pieczenie',
'smażenie']}
>>> del D['jajka']                                              # Usunięcie
wpisu
>>> D
{'mielonka': 2, 'szynka': ['grillowanie', 'pieczenie', 'smażenie']}
>>> D['lunch'] = 'Bekon'                                         # Dodanie
nowego wpisu
>>> D
{'lunch': 'Bekon', 'mielonka': 2, 'szynka': ['grillowanie', 'pieczenie',
'smażenie']}
```

Tak jak w przypadku listy, przypisanie elementu do istniejącego indeksu w słowniku również zmienia jego wartość. Jednak w przeciwieństwie do list przypisanie *nowego* klucza słownika powoduje utworzenie nowego wpisu, tak jak w powyższym przykładzie w przypadku klucza 'lunch' (tego, który nie był wcześniej przypisany). Nie działa to w przypadku list, ponieważ w listach wartości można przypisywać tylko do istniejących pozycji — Python traktuje przesunięcia spoza końca listy jako błąd. Aby rozszerzyć listę, powinieneś skorzystać z innego narzędzia, na przykład metody `append` lub przypisania wycinka.

Inne metody słowników

Metody słowników udostępniają różnorodne narzędzia specyficzne dla tego typu. Na przykład metody `values` oraz `items` zwracają odpowiednio wszystkie wartości słownika oraz krotki (*klucz, wartość*); wraz z metodą `keys` są one przydatne w pętlach, które przechodzą kolejno przez wpisy słownika (zaczniemy omawiać przykłady takich pętli w następnym podrozdziale). Podobnie jak w przypadku metody `keys`, w wersji 3.x dwie wspomniane wyżej metody również zwracają obiekty *iterowalne*, zatem aby od razu wyświetlić wyniki ich działania, powinieneś opakować je w wywołanie metody `list`:

```
>>> D = {'mielonka': 2, 'szynka': 1, 'jajka': 3}
>>> list(D.values( ))
[3, 2, 1]
>>> list(D.items( ))
[ ('jajka', 3), ('mielonka', 2), ('szynka', 1)]
```

W rzeczywistych programach, które zbierają dane podczas działania, przed uruchomieniem programu często nie będziesz w stanie przewidzieć, co znajdzie się w słowniku. Pobieranie nieistniejącego klucza jest zwykle błędem, ale jeżeli klucz nie istnieje, metoda `get` zwraca wartość domyślną `None` lub przekazaną wartość domyślną. Jest to prosty sposób na utworzenie

wartości domyślnej dla nieistniejącego klucza i uniknięcie błędu związanego z jego brakiem, gdy program nie może z góry przewidzieć zawartości słownika:

```
>>> D.get('mielonka')                      # Klucz istnieje
2
>>> print(D.get('tost'))                   # Nieistniejący klucz
None
>>> D.get('tost', 88)
88
```

Metoda `update` zapewnia słownikom coś podobnego do konkatenacji, z tym że kolejność elementów nie jest tutaj zachowywana (jak pamiętasz, w przypadku słowników kolejność elementów nie ma żadnego znaczenia). Wywołanie tej metody łączy klucze i wartości danego słownika z innym słownikiem i po prostu ślepo nadpisuje odpowiednie wartości w przypadku kolizji kluczów.

```
>>> D
{'jajka': 3, 'mielonka': 2, 'szynka': 1}
>>> D = {'tost':4, 'ciastko':5}           # Mmm, zapowiada się tutaj
pyszna jajecznica
>>> D.update(D2)
>>> D
{'jajka': 3, 'ciastko': 5, 'tost': 4, 'mielonka': 2, 'szynka': 1}
```

Zwróć uwagę, jak pomieszana jest kolejność kluczy w ostatnim wyniku — tak właśnie działają słowniki. Metoda `pop` usuwa klucz ze słownika i zwraca jego wartość. Przypomina ona metodę `pop` listy, jednak argumentem jej wywołania jest klucz, a nie opcjonalna pozycja.

```
# Usunięcie wpisu ze słownika według klucza
>>> D
{'jajka': 3, 'ciastko': 5, 'tost': 4, 'mielonka': 2, 'szynka': 1}
>>> D.pop('ciastko')
5
>>> D.pop('tost')                         # Usunięcie i zwrócenie wartości
klucza
4
>>> D
{'jajka': 3, 'mielonka': 2, 'szynka': 1}
# Usunięcie elementu z listy według jego pozycji
>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop( )                             # Usunięcie i zwrócenie
ostatniego elementu
'dd'
```

```

>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)                                # Usunięcie elementu z
określonej pozycji
'bb'
>>> L
['aa', 'cc']

```

Słowniki zawierają również metodę `copy`; powrócimy do niej w rozdziale 9., ponieważ jest ona sposobem pozwalającym na uniknięcie potencjalnych efektów ubocznych wynikających ze współdzielonych referencji do tego samego słownika. Tak naprawdę słowniki mają o wiele więcej metod, niż zaprezentowano w tabeli 8.2. Pełną ich listę można znaleźć w dokumentacji Pythona, wynikach działania `dir` i `help` oraz innych źródłach.



Kolejność elementów w Twoich słownikach może być zupełnie inna — nie powinieneś się zatem niepokoić, gdy zawartość Twoich słowników będzie wyświetlana w zupełnie innej kolejności niż pokazana tutaj w przykładach. Jak już wspominaliśmy, kolejność kluczy jest praktycznie losowa i może różnić się w zależności od wersji Pythona, platformy i sesji interaktywnej (i całkiem możliwe, że również od dnia tygodnia i fazy Księżyca!).

Większość przykładów ze słownikami w tej książce odzwierciedla kolejność kluczy w wersji 3.3, ale będzie ona inna zarówno w wersji 3.0, jak w wersjach wcześniejszych. Kolejność kluczy w Pythonie może się różnić, ale nie powinieneś się tym przejmować: słowniki są przetwarzane według kluczy, a nie pozycji. Programy nie powinny polegać na arbitralnej kolejności kluczy w słownikach, nawet jeżeli tak to wygląda w książkach.

Istnieją standardowe typy rozszerzeń w standardowej bibliotece Pythona, które utrzymują klucze słownika w określonym porządku — zobacz słowniki typu `OrderedDict` w module `collections` — ale są to hybrydy, których nowa funkcjonalność wymaga dodatkowego miejsca w pamięci i dodatkowych zasobów do przetwarzania, i które nie są prawdziwymi słownikami. W skrócie możemy powiedzieć, że w słownikach typu `OrderedDict` klucze są przechowywane nadmiarowo w tablicy wiązanej (ang. *linked list*), dzięki czemu możemy na nich wykonywać operacje sekwencyjne.

Jak zobaczymy w rozdziale 9., moduł `collections` implementuje krotki typu `namedtuple`, które umożliwiają dostęp do elementów krotki zarówno według nazwy atrybutu, jak i pozycji — jest to rodzaj hybrydy krotki, klasy i słownika, która wymaga dodatkowego przetwarzania i w żadnym razie nie jest podstawowym typem obiektu. Więcej szczegółowych informacji na temat tych i innych typów rozszerzeń znajdziesz w dokumentacji Pythona.

Przykład – baza danych o filmach

Spójrzmy na bardziej realistyczny przykład słownika. Mając na względzie nazwę naszego języka programowania, poniższy przykład tworzy w pamięci komputera prostą bazę danych filmów Monty Pythona jako tabelę, która odwzorowuje *datę premiery filmów* (klucze) na tytuły filmów (*wartości*). Dzięki takiemu rozwiążaniu możemy pobierać nazwy filmów według indeksów reprezentujących ich daty premiery:

```

>>> table = {'1975': 'Monty Python i Święty Graal',          #
klucz:wartość

```

```

...
    '1979': 'Żywot Briana',
...
    '1983': 'Sens życia według Monty Pythona'}
```

>>>

```

>>> year = '1983'
>>> movie = table[year]                                # słownik[klucz] => wartość
>>> movie
'Sens życia według Monty Pythona'
>>> for year in table:                               # To samo co: for year in
    table.keys()
... print(year + '\t' + table[year])
...
1979 Żywot Briana
1975 Monty Python i Święty Graal
1983 Sens życia według Monty Pythona
```

W ostatnim poleceniu wykorzystano pętlę `for`, o której wspominaliśmy już w rozdziale 4., ale nie mieliśmy okazji omawiać bardziej szczegółowo. Jeżeli nie wiesz, jak działa pętla `for`, to na razie powinna Ci wystarczyć informacja, że polecenie to przechodzi po kolejni przez wszystkie klucze tabeli i wyświetla rozzieloną tabulatorami listę kluczy i ich wartości. Więcej szczegółowych informacji na temat pętli `for` znajdziesz w rozdziale 13.

Ponieważ słowniki nie są sekwencjami, nie można za pomocą pętli `for` iterować po nich w sposób bezpośredni, tak jak to robiliśmy w przypadku łańcuchów znaków oraz list. Jeżeli jednak chcesz przechodzić przez kolejne elementy słownika, jest to łatwe — aby to zrobić, wystarczy wywołać metodę `keys` zwracającą listę wszystkich kluczy słownika, po której można iterować za pomocą pętli `for`. W razie potrzeby możesz wewnątrz pętli indeksować słownik kluczem i pobierać kolejne wartości — tak jak to zrobiliśmy w przykładzie powyżej.

Tak naprawdę Python w większości pętli `for` pozwala również przechodzić listę kluczy słownika bez wywoływania metody `keys`. Dla dowolnego słownika `D` wyrażenie `for key in D:` działa tak samo jak pełne wyrażenie `for key in D.keys():`. To tak naprawdę kolejny przypadek wspomnianych wcześniej iteratorów, które pozwalają na działanie operatora przynależności `in` również na słownikach (więcej szczegółowych informacji na temat iteratorów znajdziesz w dalszej części książki).

Przykład — mapowanie wartości na klucze

Zwróć uwagę, w jaki sposób poprzednia tabela odwzorowuje daty premiery na tytuły filmów, ale nie odwrotnie. Jeżeli chcesz dokonać innego odwzorowania — na przykład tytułów na daty premiery — możesz albo zakodować słownik w inny sposób, albo użyć metod takich jak `items`, które dają przeszukiwalne sekwencje, chociaż korzystanie z nich w pełni wymaga nieco większego zasobu wiedzy, niż udało nam się do tej pory przekazać:

```

>>> table = {'Monty Python i Święty Graal': '1975',          # klucz=>wartość
            (tytuł=>data premiery)
...
    'Żywot Briana ': '1979',
...
    'Sens życia według Monty Pythona ': '1983'}
```

>>>

```

>>> table[' Monty Python i Święty Graal']

'1975'

>>> list(table.items())
# wartość=>klucz
(data premiery=>tytuł)

[(' Sens życia według Monty Pythona ', '1983'), (' Monty Python i Święty
Graal', '1975'), (' Żywot Briana ', '1979')]

>>> [title for (title, year) in table.items() if year == '1975']

[' Monty Python i Święty Graal']

```

Ostatnie polecenie tutaj jest po części podglądem składni złożień, wprowadzonej w rozdziale 4. i omówionej bardziej szczegółowo w rozdziale 14. Mówiąc w skrócie, przeglądamy tutaj krótki (*klucz, wartość*) słownika zwrócone przez metodę `items`, wybierając klucze mające określoną wartość. Efektem netto jest indeksowanie `wstecz`, od wartości do klucza, zamiast od klucza do wartości — jest to przydatne, jeżeli chcesz przechowywać dane tylko raz i niezbyt często używać odwzorowania wstecznego (przeszukiwanie takich sekwencji jest na ogół znacznie wolniejsze niż bezpośrednie użycie indeksu według klucza).

W rzeczywistości, chociaż słowniki z natury dokonują jednokierunkowego odwzorowania kluczy na wartości, istnieje wiele sposobów, które z użyciem niewielkiej ilości dodatkowego kodu pozwalają na wsteczne odwzorowywanie wartości z powrotem na klucze:

```

>>> K = ' Monty Python i Święty Graal '

>>> table[K]
# klucz=>wartość
(normalny sposób użycia)

'1975'

>>> V = '1975'

>>> [key for (key, value) in table.items() if value == V]      #
wartość=>klucz

['Holy Grail']

>>> [key for key in table.keys() if table[key] == V]          # jw.

['Holy Grail']

```

Zwróci uwagę, że dwa ostatnie polecenia zwracają *listę* tytułów: w słownikach do każdego klucza przypisana jest tylko *jedna* wartość, ale *wiele* kluczy może mieć przypisaną tę samą wartość. Inaczej mówiąc, dana wartość może być przechowywana pod wieloma kluczami (czyli mamy tutaj wiele kluczy na jedną wartość), a wartością może być sama kolekcja (obsługująca wiele wartości na klucz). Aby uzyskać więcej informacji na ten temat, powinieneś również zwrócić uwagę na funkcję odwracania słownika w przykładzie `mapattrs.py` z rozdziału 32. — jest to jednak kod, który z pewnością wykracza poza ramy tego rozdziału, i dlatego nie będziemy go tutaj zamieszczać, natomiast w tym rozdziale omówimy jeszcze kilka podstawowych zagadnień związanych ze słownikami.

Uwagi na temat korzystania ze słowników

Słowniki są dość proste w użyciu, gdy się je zrozumie, jednak zawsze warto pamiętać o kilku wskazówkach i uwagach związanych z ich stosowaniem:

- **Na słownikach nie działają operacje sekwencyjne.** Słowniki są odwzorowaniami, a nie sekwencjami. Ponieważ w przypadku elementów słownika nie ma mowy o kolejności i

uporządkowaniu, nie mają tu zastosowania pewne operacje, takie jak konkatenacja (uporządkowane łączenie) czy wycinki (ekstrakcja przylegającego fragmentu). Tak naprawdę, jeżeli spróbujemy zrobić coś takiego, Python zwróci błąd.

- **Przypisanie do nowych indeksów dodaje wpisy.** Klucze można tworzyć albo przy zapisywaniu literala słownika (w tym przypadku są one osadzone w samym literale), albo kiedy przypisujemy wartości do nowych kluczy istniejącego obiektu słownika. Rezultat będzie taki sam.
- **Klucze nie zawsze muszą być łańcuchami znaków.** W naszych przykładach kluczami były właśnie łańcuchy, ale ich rolę mogą równie dobrze spełniać dowolne obiekty *niemutowalne*. Można na przykład jako klucze wykorzystywać liczby całkowite, co sprawia, że słownik zaczyna przypominać listę (przynajmniej w czasie indeksowania). Czasami w tej roli wykorzystywane są również krotki, pozwalające na tworzenie złożonych wartości kluczów, np. składających się z dat i adresów IP. Jako klucze można wykorzystać również obiekty instancji klas definiowanych przez użytkownika (które zostaną omówione w szóstej części książki), o ile będą posiadały odpowiednie metody protokołów. W uproszczeniu: muszą one „poinformować” Pythona, że ich wartości są „haszowalne” i dlatego nie będą się zmieniać, ponieważ w przeciwnym razie byłyby bezużyteczne i nie mogłyby być wykorzystywane jako stałe klucze słownika. Obiekty mutowalne, takie jak listy, zbiory i inne słowniki, nie mogą być kluczami, ale są dozwolone jako wartości.

Wykorzystywanie słowników do symulowania elastycznych list – liczby całkowite jako klucze

Ostatni punkt listy z poprzedniej sekcji jest na tyle ważny, że warto go poprzeć kilkoma przykładami. Kiedy używamy list, nie możemy przypisać elementu do pozycji przesunięcia znajdującej się poza końcem tej listy.

```
>>> L = []
>>> L[99] = 'mielonka'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Choć można wykorzystać powtórzenie do wstępnego utworzenia wystarczająco dużej listy (na przykład `[0]*100`), podobny efekt można również uzyskać za pomocą słowników, które nie wymagają takiej alokacji miejsca. Dzięki użyciu kluczy w postaci liczb całkowitych słowniki mogą emulować listy rosnące dzięki przypisywaniu kolejnych elementów do nowych pozycji przesunięcia.

```
>>> D = {}
>>> D[99] = 'mielonka'
>>> D[99]
'mielonka'
>>> D
{99: 'mielonka'}
```

D wygląda na listę stwiercanową, jednak tak naprawdę jest tylko słownikiem z jednym wpisem. Wartością klucza 99 jest łańcuch znaków 'mielonka'. Dostęp do tej struktury można uzyskać za pomocą wartości przesunięcia, sprawdzając w razie potrzeby za pomocą `get` lub `in`, czy dany klucz istnieje — dzięki takiemu rozwiązaniu nie musisz z góry alokować miejsca na wszystkie elementy, jakie kiedykolwiek będą się na takiej „liście” pojawiały w przeszłości. W takich zastosowaniach słowniki są bardziej elastycznymi odpowiednikami list.

Innym przykładem może być zastosowanie kluczy w postaci liczb całkowitych w naszej bazie danych o filmach, dzięki czemu daty premier poszczególnych filmów nie będą musiały być umieszczane w cudzysłowie, choć będzie to kosztem pewnej elastyczności (klucze nie mogą zawierać znaków innych niż cyfry):

```
>>> table = {1975: 'Monty Python i Święty Graal',
...           1979: 'Żywot Briana',                      # Klucze są liczbami
...           całkowitymi, a nie ciągami znaków
...           1983: 'Sens życia według Monty Pythona'}
>>> table[1975]
'Holy Grail'
>>> list(table.items())
[(1979, 'Żywot Briana'), (1983, 'Sens życia według Monty Pythona'), (1975,
'Monty Python i Święty Graal')]
```

Wykorzystywanie słowników z rzadkimi strukturami danych — krotki jako klucze

W podobny sposób klucze słowników można wykorzystać do implementowania *rzadkich struktur danych* (ang. *sparse data structures*) — na przykład tablic wielowymiarowych, w których jedynie kilka pozycji zawiera wartości.

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4                      # średnik rozdziela poszczególne
polecenia; zobacz rozdział 10.
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

W powyższym kodzie słownik wykorzystaliśmy do reprezentacji tablicy trójwymiarowej, która jest prawie pusta — zawiera jedynie dwie pozycje: (2, 3, 4) oraz (7, 8, 9). Klucze są *krotkami* przechowującymi współrzędne niepustych wpisów. Zamiast alokować dużą i w większości pustą macierz trójwymiarową, możemy skorzystać z prostego, dwuelementowego słownika. W takim układzie próba dostępu do pustego wpisu kończy się błędem nieistniejącego klucza, ponieważ wpisy takie nie są fizycznie przechowywane.

```
>>> Matrix[(2,3,6)]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    KeyError: (2, 3, 6)
```

Unikanie błędów z brakującymi kluczami

Błedy wynikające z prób dostępu do nieistniejących kluczy w macierzach rzadkich zdarzają się bardzo często, jednak zazwyczaj wolelibyśmy, aby nie kończyły one działania całego programu. Istnieją przynajmniej trzy sposoby na zwracanie wartości domyślnych zamiast generowania błędów tego typu. Możemy z góry sprawdzać istnienie kluczy za pomocą instrukcji `if`, wykorzystać instrukcję `try` do przechwycenia i obsługi błędu albo po prostu wykorzystać zaprezentowaną wcześniej metodę `get` słownika do podania wartości domyślnej dla nieistniejących kluczy. Zwrócić szczególną uwagę na pierwsze dwa wyrażenia — będziemy je bardziej szczegółowo omawiać w rozdziale 10.:

```
>>> if (2,3,6) in Matrix:      # Sprawdzanie kluczy przed pobraniem
...     print (Matrix[(2,3,6)])  # Więcej w rozdziałach 10.i 12., gdzie
będziemy omawiać konstrukcję if/else
... else:
...     print 0
...
0
>>> try:
...     print (Matrix[(2,3,6)])  # Próba indeksowania
... except KeyError:           # Przechwycenie i obsługa błędu
...     print 0                 # Więcej w rozdziałach 10. i 34., gdzie
będziemy omawiać try/except
...
0
>>> Matrix.get((2,3,4), 0)    # Istnieje – pobieramy i zwracamy wartość
88
>>> Matrix.get((2,3,6), 0)    # Nie istnieje – używamy wartości domyślnej
0
```

Jak widać w powyższych przykładach, metoda `get` jest najbardziej zwięzła pod względem wymagań dotyczących kodowania, ale instrukcje `if` i `try` mają znacznie bardziej ogólny zakres zastosowań; więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 10.

Zagnieżdżanie słowników

Jak widać, słowniki mogą w Pythonie odgrywać różne role. Mogą zastępować struktury wyszukujące dane (ponieważ indeksowanie według kluczy jest operacją wyszukiwania), a także reprezentować różne typy ustrukturyzowanych informacji. Na przykład słowniki są jednym z wielu sposobów opisywania właściwości różnych elementów w programie; oznacza to, że mogą pełnić tę samą rolę, co „rekordy” lub „struktury” w innych językach.

Poniższy przykład wypełnia słownik opisujący hipotetyczną osobę, przypisując nowe klucze w miarę upływu czasu:

```
>>> rec = {}
>>> rec['name'] = 'Adam'
>>> rec['age'] = 40.5
>>> rec['job'] = 'deweloper/menedżer'
```

```
>>>  
>>> print rec['name']  
Adam
```

Dzięki możliwości zagnieżdżania wbudowane typy danych Pythona pozwalają z łatwością reprezentować informacje *ustrukturyzowane*. Poniższy przykład ponownie wykorzystuje słownik do zapamiętania właściwości obiektu, jednak w tym przypadku wszystko kodowane jest za jednym razem (a nie przypisywane pojedynczo do poszczególnych kluczy). Wartości ustrukturyzowanych właściwości obiektu zostają zapisane przy użyciu zagnieżdżonej listy oraz słownika.

```
>>> rec = {'name': 'Adam',
...         'jobs': ['deweloper', 'menedżer'],
...         'web': 'www.bobs.org/~Adam',
...         'home': {'state': 'Przepracowany', 'zip':12345}}
```

Aby pobrać komponenty zagnieżdżonych obiektów, należy połączyć ze sobą operacje indeksowania.

```
>>> rec['name']
'Adam'
>>> rec['jobs']
['deweloper', 'menedżer']
>>> rec['jobs'][1]
'menedżer'
>>> rec['home']['zip']
12345
```

Choć z części VI dowiesz się, że *klasy* (które grupują zarówno dane, jak i logikę) mogą się lepiej sprawdzać w roli rekordów, słowniki są łatwym w użyciu narzędziem dla mniejszych wymagających zadań. Więcej szczegółowych informacji o sposobach reprezentacji rekordów znajdziesz w ramce „Warto pamiętać — słowniki kontra listy” w dalszej części tego rozdziału, a także w omówieniu krotek w rozdziale 9, i klas w rozdziale 27.

Zauważ też, że chociaż skupiliśmy się tutaj na pojedynczym „rekordzie” z zagnieździonymi danymi, nie ma powodu, dla którego nie moglibyśmy zagnieździć samego rekordu w większej kolekcji spełniającej rolę *bazy danych*, zakodowanej jako lista lub słownik; w praktyce rolę kontenera najwyższego poziomu często spełnia plik zewnętrzny lub formalny interfejs bazy danych (poniższe fragmenty kodu po uruchomieniu z poziomu wiersza poleceń sesji interaktywnej wyświetla dwuelementową listę zadań Adama):

```
db = []
db.append(rec)                      # "Baza danych" w postaci listy
db.append(other)
db[0]['jobs']
db = {}
db['adam'] = rec                   # "Baza danych" w postaci słownika
```

```
db['kasia'] = other  
db['adam']['jobs']
```

W dalszej części książki spotkamy takie narzędzia jak moduł `shelve`, który działa w bardzo podobny sposób, z tym że automatycznie mapuje obiekty do i z plików, dzięki czemu są zachowywane na stałe (więcej szczegółowych informacji na ten temat znajdziesz w ramce „Warto pamiętać — interfejsy słowników” w dalszej części tego rozdziału).

Inne sposoby tworzenia słowników

Warto pamiętać, że dzięki popularności słowników z czasem powstało wiele sposobów ich tworzenia. W Pythonie 2.3 i nowszych mamy do dyspozycji konstruktor `dict`, któremu przekazuje się poszczególne elementy tworzonego słownika w parametrach słownikowych lub w postaci listy par klucz/wartość. W wyniku tego typu wywołanie konstruktora daje ten sam efekt co literał słownikowy i przypisania do kluczy.

```
{'name': 'Adam', 'age': 40}      # Tradycyjne wyrażenie literała słownika  
D = {}                          # Dynamiczne przypisanie do kluczy  
  
D['name'] = 'Adam'  
  
D['age'] = 40  
  
dict(name='Adam', age=40)        # Wywoływanie konstruktora dict z argumentami  
w postaci kluczy i wartości  
  
dict([('name', 'Adam'), ('age', 40)]) # Wywoływanie konstruktora dict z  
argumentami w postaci krotek
```

Wszystkie powyższe wyrażenia tworzą taki sam słownik zawierający dwa klucze, ale każda z tych form jest przydatna w nieco innych okolicznościach:

- pierwsza z nich przydaje się w sytuacjach, kiedy jesteś w stanie statycznie zadeklarować całą zawartość słownika;
- druga jest użyteczna w sytuacji, gdy słownik chcesz tworzyć dynamicznie, po jednym elemencie;
- trzecia forma wymaga mniej pisania od pierwszej, ale wymaga, aby klucze słownika były ciągami znaków;
- ostatnia forma jest najbardziej użyteczna w przypadku, gdy posiadamy listę par klucz/wartość, wygenerowaną dynamicznie w innej części programu.

Argumenty z kluczami spotkaliśmy już wcześniej w tym rozdziale, przy okazji omawiania sortowania. Trzecia forma tworzenia słownika stała się szczególnie popularna w Pythonie, ponieważ wykorzystuje najprostszą składnię (dzięki czemu prowokuje mniejszą ilość błędów). Jak sugerowałem w tabeli 8.2, ostatnia forma przedstawiona na powyższym listingu jest również powszechnie stosowana w połączeniu z funkcją `zip`, pozwalając na połączenie osobnych list kluczy i wartości, generowanych dynamicznie po uruchomieniu programu (na przykład pobieranych z określonych kolumn z pliku danych):

```
dict(zip(keylist, valueslist))          # połączone krotki zawierające  
pary klucz/wartość
```

Więcej przykładów użycia tej formy znajdziesz w następnej sekcji. Jeżeli wszystkie klucze mają początkowo mieć przypisaną tę samą wartość, słownik można zbudować z użyciem jeszcze jednej specjalnej formy wyrażenia — wystarczy po prostu przekazać listę kluczy i wartość początkową dla wszystkich kluczy jednocześnie (wartość domyślna to `None`):

```
>>> dict.fromkeys(['a', 'b'], 0)
```

```
{'a': 0, 'b': 0}
```

Na tym etapie poznawania Pythona większości programistów wystarczają literał i przypisania według klucza, ale zdecydowanie warto również znać zastosowania innych form tworzenia słowników, które są przydatne podczas tworzenia praktycznych, elastycznych i wydajnych programów w języku Python.

Przykłady prezentowane w powyższym listingu działają w Pythonie 2.x i 3.x, ale istnieje jeszcze jeden sposób tworzenia słowników, dostępny wyłącznie w wersjach 3.x i 2.7: *wyrażenia słowników składanych* (ang. *dictionary comprehension expressions*). Szczegóły zastosowania tej techniki są opisane w następnej, ostatniej już sekcji tego rozdziału.

Warto pamiętać — słowniki kontra listy

Mając na względzie wszystkie obiekty z arsenalu podstawowych typów Pythona, niektórzy mniej doświadczeni użytkownicy mogą się zastanawiać nad wyborem między listami a słownikami. W skrócie możemy powiedzieć, że choć oba typy są elastycznymi zbiorami innych obiektów, listy przypisują elementy do pozycji, a słowniki przypisują je do bardziej mnemonicznych kluczy. Z tego względu dane przechowywane w słownikach często są dużo wygodniejsze dla użytkowników. Na przykład struktura listy zagnieżdżonej, zaprezentowanej w trzecim wierszu tabeli 8.1, może również być użyta do reprezentowania rekordu:

```
>>> L = ['Adam', 40.5, ['dev', 'mgr']]      # „rekord” oparty na liście
>>> L[0]
'Adam'
>>> L[1]                                     # Pozycje/numery pól
40.5
>>> L[2][1]
'mgr'
```

W przypadku niektórych typów danych dostęp do elementów listy według pozycji jest bardzo użytecznym rozwiązaniem — z pewnością sprawdzi się w przypadku listy pracowników w firmie, plików w katalogu czy macierzy numerycznych. Ale bardziej symboliczny zapis takiego rekordu może być lepiej zakodowany jako słownik (tak jak w drugim wierszu tabeli 8.2, z nazwami zastępującymi pozycje pól — jest to podobne do zapisu, jakiego używaliśmy w rozdziale 4.):

```
>>> D = {'name': 'Adam', 'age': 40.5, 'jobs': ['dev', 'mgr']}
>>> D['name']
'Adam'
>>> D['age']                                    # "rekord" oparty na słowniku
40.5
>>> D['jobs'][1]                                # Nazwy znaczą więcej niż liczby
'mgr'
```

Dla odmiany poniżej przedstawiamy ten sam rekord zakodowany z argumentami w postaci par klucz/wartość, które niektórym użytkownikom mogą wydawać się jeszcze bardziej czytelne:

```
>>> D = dict(name='Adam', age=40.5, jobs=['dev', 'mgr'])
```

```

>>> D['name']
'Adam'
>>> D['jobs'].remove('mgr')
>>> D
{'jobs': ['dev'], 'age': 40.5, 'name': 'Adam'}
W praktyce słowniki najlepiej sprawdzają się w przypadku danych z oznaczonymi komponentami, a także struktur, które mogą korzystać z szybkich, bezpośrednich wyszukiwań według nazwy zamiast wolniejszego wyszukiwania liniowego. Jak widzieliśmy, słowniki mogą być również lepsze w przypadku rzadkich kolekcji i kolekcji rosnących na dowolnych pozycjach.
Użytkownicy Pythona mają również dostęp do zbiorów, które badaliśmy w rozdziale 5. Zbiory są bardzo podobne do kluczy słownika bezwartościowego; nie odwzorowują kluczy na wartości, ale często mogą być używane jak słowniki do szybkiego sprawdzania, gdy nie ma powiązanych wartości, szczególnie w procedurach wyszukiwania:

```

```

>>> D = {}
>>> D['state1'] = True # Słownik odwiedzonych stanów
>>> 'state1' in D
True
>>> S = set()
>>> S.add('state1') # To samo, ale za pomocą zbioru
>>> 'state1' in S
True

```

W następnym rozdziale powrócimy do tego wątku i zobaczymy, jak w tej roli w porównaniu ze słownikami wypadają krotki i krotki nazwane. Kolejny raz zajmiemy się podobnymi zagadnieniami w rozdziale 27., gdzie dowiesz się, w jaki sposób do realizacji podobnych zadań można użyć *klas zdefiniowanych przez użytkownika*, które łączą w sobie zarówno dane, jak i logikę pozwalającą na ich przetwarzanie.

Zmiany dotyczące słowników w Pythonie 3.x i 2.7

Tematyka słowników omawiana w tym rozdziale koncentrowała się dotychczas głównie na podstawowych zagadnieniach wspólnych dla wszystkich wydań Pythona, ale mechanizm działania słowników uległ zmianom w wersji 3.x. Jeżeli używasz kodu Pythona dla wersji 2.x, możesz natknąć się na niektóre narzędzia słownikowe, które albo zachowują się inaczej, albo są niedostępne w wersji 3.x. Co więcej, programiści, którzy pracują wyłącznie w wersji 3.x, mają dostęp do nowych mechanizmów słownikowych niedostępnych w wydaniach z serii 2.x (z wyjątkiem dwóch modułów przeniesionych do wersji 2.x).

Do nowości w mechanizmach obsługi słowników w wersji 3.x zaliczamy:

- obsługę nowego wyrażenia dla *słowników składanych*, będącego bliskim kuzynem list i zbiorów składanych;
- metody `D.keys`, `D.values` i `D.items` zamiast list zwracają widoki podobne do zbiorów, będące obiektami iterowalnymi;
- używanie posortowanych kluczy wymaga zastosowania nieco innych technik programowania, co jest konsekwencją zmian opisanych w poprzednim punkcie;

- brak bezpośredniej obsługi względnych porównań wielkości słowników, wielkość słowników porównujemy ręcznie;
- brak metody `D.has_key`, zamiast tego stosujemy test przynależności z operatorem `in`.

W rezultacie dzięki nowym rozwiązaniom przeniesionym z wersji 3.x (ang. *backport*) słowniki w wersji 2.7 (ale bez wcześniejszych wersji 2.x):

- Obsługują wyrażenia *słowników składanych*, opisane w pierwszym punkcie poprzedniej listy (bezpośredni backport z wersji 3.x).
- Obsługują widoki opisane w drugim punkcie poprzedniej listy (iterowalne widoki podobne do zbiorów), ale wykorzystują do tego celu specjalne metody `D.viewkeys`, `D.viewvalues` i `D.viewitems`; pozostałe metody zwracają listy jak poprzednio.

Z powodu takiego nakładania się funkcjonalności część materiału w tej sekcji dotyczy zarówno wersji 3.x, jak i 2.7, ale ze względu na swoje pochodzenie jest tu przedstawiona w kontekście rozszerzeń wersji 3.x. Mając to na uwadze, zobaczymy, co nowego pojawiło się w słownikach w wersjach 3.x i 2.7.

Słowniki składane w wersjach 3.x i 2.7

Jak wspominaliśmy w poprzednim punkcie, w Pythonie 3.x i 2.7 słowniki mogą być tworzone z użyciem nowej konstrukcji: wyrażenia słowników składanych (ang. *dictionary comprehension*). Podobnie jak wyrażenia zbiorów składanych omówione w rozdziale 5., słowniki składane są dostępne wyłącznie w wersjach 3.x i 2.7 (nie można ich używać w wersjach 2.6 i wcześniejszych). Podobnie jak dostępne od dawna listy składane, których mieliśmy okazję używać w rozdziale 4. i nieco wcześniej w tym rozdziale, słowniki składane wykorzystują wewnętrzną pętlę, która zbiera w każdej iteracji wyniki wyrażeń dla kolejnej pary klucz/wartość i wykorzystuje je do wypełnienia nowego słownika wynikowego. Zmienna pętli pozwala na wykorzystywanie wartości pośrednich z każdej iteracji.

Pozwalającym to zilustrować standardowym sposobem dynamicznej inicjalizacji słownika zarówno w wersji 2.x, jak i 3.x jest połączenie jego kluczy i wartości za pomocą funkcji `zip` i przekazanie wyniku tej operacji do wywołania konstruktora `dict`. Wbudowana funkcja `zip` jest mechanizmem, który pozwala nam w taki sposób zbudować słownik z list kluczy i wartości — jeżeli nie możesz z góry przewidzieć zestawu kluczy i wartości przed uruchomieniem programu, zawsze możesz zbudować je później jako listy i połączyć razem w nowy słownik. Działanie funkcji `zip` przeanalizujemy szczegółowo w rozdziałach 13. i 14.; w wersji 3.x jest ona obiektem iterowalnym, zatem aby wyświetlić wyniki jej działania, musimy ją opakować w wywołanie funkcji `list`, tak jak to zostało pokazane w przykładzie poniżej. Nie zmienia to jednak w niczym faktu, że podstawowy sposób użycia tej funkcji jest bardzo prosty:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))          # Połączenie kluczy i
wartości
[('a', 1), ('b', 2), ('c', 3)]
>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))      # Utworzenie słownika z
wyniku funkcji zip
>>> D
{'b': 2, 'c': 3, 'a': 1}
```

W Pythonie 3.x i 2.7 możemy uzyskać ten sam efekt, wykorzystując wyrażenie słownika składanego. Poniższy listing przedstawia sposób utworzenia nowego słownika na podstawie wyniku działania funkcji `zip`, do której przekazujemy listy kluczy i wartości (kod języka Python odczytujemy niemal tak samo jak opis w języku naturalnym, choć z nieco większą dozą sformalizowania):

```
>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
```

```
>>> D  
{'b': 2, 'c': 3, 'a': 1}
```

W tym przypadku utworzenie słownika składanego wymagało napisania nieco większej ilości kodu, ale wyrażenia te są znacznie bardziej ogólne, niż mógłby sugerować ten przykład: między innymi można za ich pomocą zbudować słownik w oparciu o jeden strumień danych, a zarówno klucze, jak i wartości, mogą być automatycznie generowane w wyrażeniach:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}           # Lub: range(1, 5)  
>>> D  
{1: 1, 2: 4, 3: 9, 4: 16}  
>>> D = {c: c * 4 for c in 'JAJKO'}                 # Iteracja po dowolnym  
iteratorze  
>>> D  
{'A': 'AAAA', 'K': 'KKKK', 'J': 'JJJJ', 'O': '0000'}  
>>> D = {c.lower(): c + '!' for c in ['MIELONKA', 'JAJKA', 'SZYNKA']}  
>>> D  
{'szynka': 'SZYNKA!', 'jajka': 'JAJKA!', 'mielonka': 'MIELONKA!'}
```

Słowniki składane są również przydatne do inicjalizacji słowników za pomocą list kluczów, podobnie jak robi to metoda `fromkeys`, którą spotkaliśmy w poprzednim podrozdziale:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)           # Inicjalizacja słownika za  
pomocą listy kluczów  
>>> D  
{'b': 0, 'c': 0, 'a': 0 }  
>>> D = {k:0 for k in ['a', 'b', 'c']}             # To samo, ale z użyciem  
słownika składanego  
>>> D  
{'b': 0, 'c': 0, 'a': 0 }  
>>> D = dict.fromkeys('spam')                      # Inne iteratory, użycie  
wartości domyślnej  
>>> D  
{'s': None, 'p': None, 'a': None, 'm': None}  
>>> D = {k: None for k in 'spam'}  
>>> D  
{'s': None, 'p': None, 'a': None, 'm': None}
```

Podobnie jak inne wyrażenia tego typu, słowniki składane pozwalają na zastosowanie dodatkowej składni, której tutaj nie pokazaliśmy, takiej jak na przykład zagnieżdżone pętle czy klauzule `if`. Niestety, aby w pełni zrozumieć wyrażenia słowników składanych, musielibyśmy szczegółowo omówić szereg innych zagadnień związanych z instrukcjami iteracyjnymi i koncepcją iteratorów w Pythonie, a w tym miejscu książka nie mamy jeszcze wystarczającej wiedzy, aby w pełni przeanalizować tę tematykę. Wszelkie typy wyrażeń składanych (listy, zbiory, słowniki składane i generatorы) omówimy szczegółowo w rozdziałach 14. i 20., więc póki

co dalszą analizę odkładamy na później. W rozdziale 13. przy okazji omawiania pętli wróćmy jeszcze do wbudowanej funkcji `zip`, której używaliśmy w tej sekcji.

Widoki słowników w wersji 3.x (oraz wersji 2.7 przy użyciu nowych metod)

W Pythonie 3.x metody `keys`, `values` i `items` słowników zwracają *obiekty widoków*, natomiast w wersjach 2.x zwracane są listy. Taka funkcjonalność dostępna jest również w Pythonie 2.7, ale pod postacią specjalnych, odrębnych metod wymienionych wcześniej w tym rozdziale (normalne metody 2.7 nadal zwracają proste listy, aby uniknąć problemów z uruchamianiem istniejącego kodu 2.x); z tego powodu dla uproszczenia będziemy mówić, że obiekty widoków to funkcjonalność wersji 3.x.

Obiekty widoków są *iteratorami*, czyli obiektami, które generują wyniki po jednym elemencie na raz, zamiast zwracać całą listę wyników jednocześnie. Widoki słowników są obiektami iterowalnymi, a dodatkowo są ściśle zintegrowane ze słownikami, z których powstały: zachowują oryginalną kolejność elementów słownika, odzwierciedlając zmiany wprowadzane w słowniku i mogą obsługiwać operacje na zbiorach. Z drugiej strony, ponieważ nie są listami, nie obsługują bezpośrednio takich operacji jak indeksowanie czy sortowanie list, a do tego nie wyświetlają bezpośrednio swoich elementów jako normalnej listy (począwszy od wersji 3.1, możemy wyświetlić ich elementy, ale nie w postaci listy; nadal występują tutaj znaczne rozbieżności z wersją 2.x).

W rozdziale 14. bardziej formalnie przeanalizujemy tematykę obiektów iterowalnych, ale na potrzeby naszych aktualnych rozważań wystarczy pamiętać, że przed wykonaniem operacji typowych dla list, lub w celu wyświetlenia wyników w całości, wyniki metod `keys`, `values` i `items` należy przekształcić na listę za pomocą wbudowanej funkcji `list`. Na przykład w Pythonie 3.3 możemy to zrobić w następujący sposób (wyniki w innych wersjach mogą się nieznacznie różnić):

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> K = D.keys()                                # w 3.x tworzy obiekt widoku, a nie listę
>>> K
dict_keys(['b', 'c', 'a'])
>>> list(K)                                     # w razie potrzeby w 3.x możemy wymusić
listę
['b', 'c', 'a']
>>> V = D.values()                               # To samo w przypadku metod values i items
>>> V
dict_values([2, 3, 1])
>>> list(V)
[2, 3, 1]
>>> D.items()
dict_items([('b', 2), ('c', 3), ('a', 1)])
>>> list(D.items())
[('b', 2), ('c', 3), ('a', 1)]
```

```
>>> K[0]                      # Operacje list nie działają, dopóki
widoki nie zostaną przekształcone
TypeError: 'dict_keys' object does not support indexing
>>> list(K)[0]
'b'
```

Nie licząc wyświetlania wyników w sesji interaktywnej, prawdopodobnie rzadko zauważysz te zmiany, ponieważ konstrukcje z użyciem pętli w Pythonie automatycznie zmuszają obiekty iterowalne do generowania jednego wyniku w każdej iteracji:

```
>>> for k in D.keys(): print(k)    # Iteratory są automatycznie używane w
pętlach
...
b
c
a
```

Dodatkowo, słowniki w 3.x są same w sobie iteratorami zwracającymi kolejne klucze i, podobnie jak w 2.x, w wielu sytuacjach nie ma konieczności jawnego wywoływania metody `keys`.

```
>>> for key in D: print(key)      # Nie ma potrzeby wywoływania metody
keys() do iteracji
...
b
c
a
```

W przeciwieństwie do list zwracanych w 2.x, widoki słowników w 3.x po utworzeniu nie są obiektami statycznymi — *dynamicznie odzwierciedlają zmiany* wprowadzane w słowniku po utworzeniu obiektu widoku:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> K = D.keys()
>>> V = D.values()
>>> list(K)                      # Widok zachowuje tę samą kolejność
elementów co słownik źródłowy
['b', 'c', 'a']
>>> list(V)
[2, 3, 1]
>>> del D['b']                  # Modyfikacja słownika w miejscu
>>> D
```

```

{'c': 3, 'a': 1}

>>> list(K)                      # Zmiana jest odzwierciedlana w bieżącym
obiekcie widoku

['c', 'a']

>>> list(V)                      # W 2.x tak się nie dzieje – lista nie
jest powiązana ze słownikiem

[3, 1]

```

Widoki słowników i zbiory

Kolejna różnica w stosunku do wyników w postaci listy z Pythona 2.x polega na tym, że w Pythonie 3.x wyniki działania metody `keys` zachowują się jak zbiory i obsługują wiele operacji typowych dla zbiorów, takich jak iloczyn czy unia zbiorów. W przypadku wyników metody `values` taka prawidłowość nie zachodzi, a w przypadku wyników działania metody `items` operacje na zbiorach będą dostępne, jeżeli pary (*klucz, wartość*) będą unikatowe i haszowalne (niemutowalne). Biorąc pod uwagę, że zbiory zachowują się jak słowniki bez wartości (i mogą być nawet kodowane w nawiasach klamrowych, takich jak słowniki w wersjach 3.x i 2.7), taka symetria jest jak najbardziej logiczna. Zgodnie z tym, o czym mówiliśmy w rozdziale 5., elementy zbiorów są nieuporządkowane, unikatowe i niemutowalne, podobnie jak klucze słownika.

Poniższy listing prezentuje zachowanie wyników działania metody `keys` w operacjach na zbiorach (kontynuacja sesji z poprzedniego podrozdziału sekcji); wyniki działania metody `values` nie zachowują się jak zbiory, ponieważ ich elementy nie zawsze muszą być unikalne lub niemutowalne:

```

>>> K, V

(dict_keys(['c', 'a']), dict_values([3, 1]))

>>> K | {'x': 4}                  # wyniki działania metody keys (i czasami
items) zachowują się jak zbiory

{'c', 'x', 'a'}

>>> V & {'x': 4}

TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'

>>> V & {'x': 4}.values()

TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'

```

W operacjach na zbiorach widoki mogą być mieszane z innymi widokami, zbiorami i słownikami; w tym kontekście słowniki są traktowane tak samo jak ich widoki `keys`:

```

>>> D = {'a':1, 'b':2, 'c':3}

>>> D.keys() & D.keys()        # Iloczyn widoków keys

{'b', 'c', 'a'}

>>> D.keys() & {'b'}           # Iloczyn widoku keys i zbioru

{'b'}

>>> D.keys() & {'b': 1}         # Iloczyn widoku keys i słownika

{'b'}

>>> D.keys() | {'b', 'c', 'd'}  # Unia widoku keys i zbioru

```

```
{'b', 'c', 'a', 'd'}
```

Widok `items` jest również obiektem zbiorowym, pod warunkiem że zawiera wyłącznie elementy niemutowalne.

```
>>> D = {'a': 1}
>>> list(D.items())
jeżeli jest haszowalny
[('a', 1)]
>>> D.items() | D.keys()          # Unia widoku items i keys
{('a', 1), 'a'}
>>> D.items() | D
keys
{('a', 1), 'a'}
>>> D.items() | {('c', 3), ('d', 4)}      # Zbiór par klucz/wartość
{('d', 4), ('a', 1), ('c', 3)}
>>> dict(D.items() | {('c', 3), ('d', 4)})    # Funkcja dict akceptuje
również zbiory iterowalne
{'c': 3, 'a': 1, 'd': 4}
```

Jeżeli potrzebujesz odświeżenia wiadomości na temat operacji na zbiorach, powinieneś zatrzymać się tutaj, bo wersja 3.x nie posiada wbudowanej funkcji `set`. A teraz na zakończenie dyskusji przedstawimy kilka dodatkowych wskazówek ułatwiających szybkie tworzenie słowników w Pythonie 3.x.

Sortowanie kluczy słowników w wersji 3.x

Po pierwsze, ponieważ w wersji 3.x metoda `keys` nie zwraca listy, tradycyjne wzorce kodowania, stosowane w wersji 2.x do przeglądania słownika według posortowanej listy kluczy, nie będą działać w wersji 3.x:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> Ks = D.keys()                  # Sortowanie obiektu widoku nie
działa!
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'
```

Aby rozwiązać ten problem, w wersji 3.x należy albo ręcznie przekonwertować listę, albo użyć wywołania funkcji `sorted` na widoku `keys` lub w samym słowniku (o funkcji `sorted` mówiliśmy już w rozdziale 4. oraz wcześniej w tym rozdziale):

```
>>> Ks = list(Ks)                  # Przekształcenie na listę i
posortowanie wyniku
>>> Ks.sort()
>>> for k in Ks: print(k, D[k])
...
...
```

```

a 1
b 2
c 3
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> Ks = D.keys()                                # Można też użyć funkcji
sorted() na kluczach
>>> for k in sorted(Ks): print(k, D[k])        # Funkcja sorted() akceptuje
dowolny obiekt iterowany
...
# sorted() zwraca listę
a 1
b 2
c 3

```

Spośród przedstawionych sposobów użycie kluczy słownika jako iteratora jest prawdopodobnie preferowanym rozwiązaniem w wersji 3.x i działa również w wersji 2.x:

```

>>> D
{'b': 2, 'c': 3, 'a': 1}                          # Jeszcze lepiej będzie
posortować sam słownik
>>> for k in sorted(D): print(k, D[k])        # Iteratory słowników zwracają
klucze
...
a 1
b 2
c 3

```

Porównywanie rozmiarów słowników nie działa w 3.x

Po drugie, w Pythonie 2.x mamy możliwość porównywania wielkości względnej z użyciem operatorów `<`, `>` itp., ale w wersji 3.x taka operacja nie jest już obsługiwana. Zachowanie to można jednak zasymulować, ręcznie porównując posortowane listy kluczy:

```
sorted(D1.items()) < sorted(D2.items())          # Odpowiednik D1 < D2 w 2.x
```

Operacje sprawdzania równości słowników (np. `D1 == D2`) nadal jednak działają w wersji 3.x. Powrócimy do tego pod koniec następnego rozdziału przy okazji szerszego omawiania zagadnień związanych z porównaniami, zatem teraz nie będziemy się w nie bardziej zagłębiać.

W wersji 3.x metoda has_key nie istnieje, niech żyje in!

Powszechnie stosowana we wcześniejszych wersjach Pythona metoda `has_key` w wersji 3.x nie jest już dostępna. Zamiast niej powinieneś używać testu przynależności `in` lub metody `get` z wartością domyślną (jednak to użycie operatora `in` jest preferowanym rozwiązaniem).

```

>>> D
{'b': 2, 'c': 3, 'a': 1}

```

```
>>> D.has_key('c')                                     # Tylko dla 2.x:  
True/False  
  
AttributeError: 'dict' object has no attribute 'has_key'  
  
>>> 'c' in D                                       # Wymagane w 3.x  
True  
  
>>> 'x' in D                                       # Preferowane  
obecnie w 2.x  
  
False  
  
>>> if 'c' in D: print('present', D['c'])          # Wybór w  
zależności od rezultatu  
  
...  
  
present 3  
  
>>> print(D.get('c'))                            # Pobieranie z  
wartością domyślną  
3  
  
>>> print(D.get('x'))  
None  
  
>>> if D.get('c') != None: print('present', D['c'])  # Inne rozwiązanie  
  
...  
present 3
```

Podsumowując, działanie słowników zmienia się zasadniczo w wersji 3.x. Jeżeli pracujesz w wersji 2.x i zależy Ci na *kompatybilności* z wersją 3.x (lub podejrzewasz, że kiedyś będzie Ci to potrzebne), poniżej znajdziesz dwie proste wskazówki:

- Po pierwsze słowniki składane można zakodować tylko w wersjach 3.x i 2.7.
 - Po drugie widoki słownika można kodować tylko w wersji 3.x, a z użyciem specjalnych metod również w wersji 2.7

Pamiętaj, że trzy opisywane nieco wcześniej techniki — funkcja `sorted`, ręczne porównania oraz operator `in` — mogą być dzisiaj używane w wersji 2.x, aby ułatwić migrację do wersji 3.x w przyszłości.

Warto pamiętać – interfeisy słowników

Poza tym, że słowniki są wygodnym sposobem przechowywania w programach informacji pogrupowanych według kluczy, w niektórych rozszerzeniach do Pythona dostępne są również interfejsy wyglądające i działające jak słowniki. Na przykład interfejs Pythona pozwalający na dostęp do plików bazy DBM, przechowujących dane w postaci par klucz/wartość, przypomina mechanizm dostępu do słownika, gdzie dane w postaci łańcuchów tekstu zapisujemy i pobieramy za pomocą indeksów w postaci kluczy:

```
import dbm # W Pythonie 2.x moduł ten nosi  
nazwę anydbm  
  
file = dbm.open("nazwa_pliku") # Łączy do pliku  
  
file['key'] = 'data' # Zapisanie danych według klucza
```

```
data = file['key']                                # Pobranie danych według klucza
```

W rozdziale 28. zobaczymy również, że w ten sposób można pobierać całe obiekty Pythona, jeżeli zastąpimy dbm w kodzie powyżej odwołaniem do modułu shelf (obiekty shelf to rodzaj bazy danych typu klucz/wartość, przechowującej trwałe obiekty Pythona, a nie tylko łańcuchy znaków). W zastosowaniach internetowych obsługa skryptów CGI w Pythonie również posiada interfejs podobny do słownika. Wywołanie cgi.FieldStorage zwraca obiekt podobny do słownika z jednym wpisem dla każdego pola formularza po stronie klienta.

```
import cgi
```

```
form = cgi.FieldStorage()                         # Analiza danych z formularza
```

```
if form.has_key('name'):
```

```
    showReply('Witaj, ' + form['name'].value)
```

Chociaż słowniki są jedynym typem reprezentującym odwzorowania w gronie typów podstawowych, wszystkie pozostałe typy podobne są na nich oparte i obsługują większość tych samych operacji. Gdy zrozumiesz zasady działania i nauczysz się korzystać z interfejsów słowników, przekonasz się, że mają one zastosowanie do wielu różnych wbudowanych narzędzi w Pythonie.

Inny przykład użycia słownika znajdziesz w rozdziale 9., gdzie będziemy się zajmować danymi w formacie JSON — neutralnym językowo formacie danych używanym w wielu bazach danych oraz powszechnie stosowanym do przesyłania danych. Słowniki, listy i ich zagnieżdżone kombinacje w Pythonie pozwalają na łatwe przekazywanie rekordów danych w tym formacie i mogą być łatwo konwertowane na i z formalnego formatu JSON za pomocą standardowego modułu biblioteki json.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy listy oraz słowniki — dwa chyba najczęściej spotykane, najbardziej elastyczne i mające największe możliwości typy kolekcji spotykanych i używanych w Pythonie. Dowiedzieliśmy się, że listy obsługują uporządkowane kolekcje dowolnych obiektów, które można dowolnie zagnieżdżać. Listy mogą również rosnąć i kurczyć się na żądanie. Słowniki są podobne, jednak przechowują elementy według kluczy, a nie ich pozycji, i nie zachowują żadnego stałego uporządkowania elementów od lewej do prawej strony. Zarówno listy, jak i słowniki są typami mutowalnymi, dlatego obsługują różnorodne operacje modyfikujące je w miejscu, jakie nie są dostępne dla łańcuchów znaków. Listy można na przykład rozszerzać za pomocą wywołań metody append, a słowniki poprzez przypisanie wartości do nowych kluczy.

W kolejnym rozdziale zakończymy nasze pogłębione omówienie podstawowych typów danych, prezentując krotki oraz pliki. Potem przejdziemy do instrukcji kodujących logikę przetwarzającą nasze obiekty, co będzie kolejnym krokiem na drodze do tworzenia kompletnych programów. Zanim jednak zajmiemy się tymi zagadnieniami, pora przejść do quizu podsumowującego rozdział.

Sprawdź swoją wiedzę — quiz

1. Podaj dwa sposoby utworzenia listy składającej się z pięciu zer (zawierającej pięć liczb całkowitych o wartości 0).
2. Podaj dwa sposoby utworzenia słownika zawierającego dwa klucze ('a' oraz 'b'), z których każdy ma przypisaną wartość 0.
3. Podaj cztery operacje modyfikujące obiekt listy w miejscu.
4. Podaj cztery operacje modyfikujące obiekt słownika w miejscu.
5. Dlaczego często lepiej jest używać słownika zamiast listy?

Sprawdź swoją wiedzę – odpowiedzi

1. Wyrażenie z literałem, takie jak `[0, 0, 0, 0, 0]`, i wyrażenie z powtórzeniem, takie jak `[0] * 5`, utworzą listę pięciu zer. W praktyce można również utworzyć taką listę za pomocą pętli, która rozpoczyna się od pustej listy i przy każdej iteracji dodaje do niej 0 — `L.append(0)`. Może tutaj zadziałać również lista składana (`[0 for i in range(5)]`), ale takie rozwiązanie wymaga większej ilości pracy, niż jest to niezbędne, aby zrealizować zadanie postawione w pytaniu.
2. Wyrażenie z literałem, takie jak `{'a': 0, 'b': 0}`, lub seria przypisań, jak `D = [0], D['a'] = 0, D['b'] = 0`, przydadzą się do utworzenia pożądanego słownika. Można również skorzystać z nowszej i prostszej do zakodowania formy ze słowami kluczowymi `dict(a=0, b=0)` lub bardziej elastycznej formy z sekwencjami klucz-wartość `dict([('a', 0), ('b', 0)])`. A także — ponieważ wszystkie klucze mają tę samą wartość — można skorzystać ze specjalnej formy `dict.fromkeys(['a', 'b'], 0)`. W wersjach 3.x i 2.7 można również użyć słownika składanego: `{k:0 for k in 'ab'}`, choć znów to już troszkę przesada.
3. Metody `append` i `extend` pozwalają rozszerzać listę w miejscu, metody `sort` i `reverse` sortują i odwracają listę, metoda `insert` wstawia element w podane miejsce, metody `remove` i `pop` usuwają element z listy po jego wartości i pozycji, instrukcja `del` usuwa element lub wycinek, natomiast instrukcje przypisania do indeksu i wycinka zastępują element lub cały fragment listy. Na potrzeby quzu wystarczy wybrać cztery odpowiedzi.
4. Słowniki zmienia się przede wszystkim poprzez przypisanie do nowego lub istniejącego klucza, co tworzy lub modyfikuje wpis klucza w tabeli. Instrukcja `del` usuwa wpis dla danego klucza, metoda słownika `update` łączy jeden słownik z drugim w miejscu, natomiast metoda `D.pop(klucz)` usuwa klucz i zwraca wartość, jaką on miał. Słowniki mają również inne, bardziej egzotyczne metody modyfikujące je w miejscu, które nie zostały wymienione w niniejszym rozdziale (na przykład `setdefault`). Więcej informacji na ten temat można znaleźć w innych materiałach źródłowych.
5. Słowniki na ogół lepiej sprawdzają się, gdy dane są oznaczone etykietami (na przykład dla rekordów z nazwami pól); listy najlepiej nadają się do kolekcji nieoznaczonych elementów (takich jak pliki w katalogu). Przeszukiwanie słownika jest zwykle szybsze niż przeszukiwanie listy, chociaż to może się różnić w zależności od programu.

[1] W praktyce niewiele list w programach przetwarzających listy tak wygląda. Znacznie częściej widuje się kod przetwarzający listy tworzone dynamicznie, w czasie działania

programu, na podstawie danych wprowadzanych przez użytkownika, zawartości plików itp. W rzeczywistości, choć opanowanie składni literałów jest niewątpliwie bardzo ważne, większość struktur danych w Pythonie jest budowana dynamicznie w czasie działania programu.

[2] Opis ten wymaga rozszerzenia, kiedy wartość i przypisywany wycinek nakładają się na siebie. Na przykład `L[2:5] = L[3:6]` zadziała dobrze, ponieważ wartość, która ma zostać wstawiona, jest pobierana, zanim po lewej stronie zostanie wykonana operacja usunięcia.

[3] W przeciwnieństwie do konkatenacji (z operatorem `+`) metoda `append` nie musi generować nowych obiektów, więc jest zazwyczaj szybsza. Metodę tę można również sprytnie symulować za pomocą odpowiedniego przypisywania do wycinków — `L[len(L):]=[X]` jest jak `L.append(X)`, natomiast `L[:0] = [X]` przypomina dodawanie elementu na początku listy. Oba rozwiązania są jednak prawdopodobnie bardziej złożone niż metody listy. Na przykład wywołanie `L.insert(0, X)` może również dołączyć element na początku listy i wydaje się zauważalnie bardziej czytelne; `L.insert(len(L), X)` także wstawia jeden obiekt na końcu listy, ale jeżeli nie lubisz pisać, równie dobrze możesz użyć wywołania `L.append(X)`!

[4] Podobnie jak to miało miejsce w przypadku list, w praktyce słowniki bardzo rzadko tworzymy za pomocą literałów — programy rzadko znają wszystkie dane przed rozpoczęciem działania i znacznie częściej tworzą je dynamicznie, pobierając od użytkowników, ładując z plików itp. Listy i słowniki tworzone są jednak na różne sposoby. Jak zobaczymy w następnym podrozdziale, słowniki zazwyczaj rozszerzamy, przypisując wartości do nowych kluczy w czasie działania programu. Takie podejście nie działa jednak w przypadku list, które rozszerzamy za pomocą metody `append`.

Rozdział 9. Krotki, pliki i wszystko inne

Niniejszy rozdział kończy nasze pogłębione omówienie podstawowych typów obiektów w Pythonie, przedstawiając *krotkę* (kolekcję innych obiektów, która nie może być zmodyfikowana) oraz *plik* (interfejs do plików zewnętrznych na komputerze). Jak zobaczymy niebawem, krotka to względnie prosty obiekt wykonujący operacje, które omówiliśmy już dla łańcuchów znaków oraz list. Obiekt pliku to powszechnie używane i w pełni funkcjonalne narzędzie do przetwarzania plików. Ponieważ pliki są tak wszechobecne w programowaniu, podstawowy przegląd zagadnień związanych z plikami zostanie uzupełniony większymi przykładami w dalszych rozdziałach książki.

Niniejszy rozdział kończy również drugą część książki, przedstawiając właściwości wspólne dla wszystkich obiektów podstawowych, z jakimi się spotkaliśmy — koncepcje równości, porównań czy kopii obiektów. Krótko omówimy również inne typy obiektów Pythona, takie jak obiekt zastępczy `None` i hybrydowe krotki typu `namedtuple`. Jak się przekonasz, choć przedstawiliśmy wszystkie najważniejsze typy wbudowane, koncepcja obiektów w Pythonie jest znacznie szersza, niż może to wynikać z dotychczasowych opisów. Zakończymy tę część książki, przyglądając się często popełnianym błędom związanym z typami obiektów i proponując kilka ćwiczeń, które pozwolą Ci na samodzielne eksperymentowanie z przedstawionymi tu koncepcjami.



Tematyka tego rozdziału to pliki, ale podobnie jak w rozdziale 7., dotyczącym ciągów, nasze spojrzenie na pliki będzie tu ograniczone do kilku podstawowych zagadnień, które powinien znać każdy programista używający Pythona, z zupełnymi nowicjuszami włącznie. *Pliki tekstowe Unicode* zostały już wcześniej przedstawione w rozdziale 4., ale odłożymy ich pełne omówienie do rozdziału 37., który możesz potraktować jako opcjonalną lub po prostu odroczoną w czasie część książki, poruszającą bardziej zaawansowane tematy.

Na potrzeby tego rozdziału przyjmiemy założenie, że wszelkie użyte tutaj pliki tekstowe zostaną zakodowane i zdekodowane zgodnie z domyślnymi ustawieniami Twojej platformy; w systemie Windows może to być UTF-8 albo ASCII, a jeszcze co innego na innej platformie (a jeżeli nie wiesz, dlaczego to ma znaczenie, prawdopodobnie nie musisz się jeszcze tym teraz przejmować). Zakładamy również, że nazwy naszych plików działają poprawnie na Twojej platformie bazowej, ale i tak na wszelki wypadek pozostaniemy przy nazwach wykorzystujących tylko znaki z podstawowego zestawu ASCII.

Jeżeli tekst i pliki w standardzie Unicode są dla Ciebie zagadnieniem krytycznym, powinieneś ponownie zajrzeć do rozdziału 4. w celu szybkiego przypomnienia sobie prezentowanych tam tematów, a następnie przeskoczyć do rozdziału 37. po opanowaniu omawianych tutaj podstawowych zagadnień dotyczących plików — tematy poruszane w tym rozdziale będą dotyczyć zarówno typowych plików tekstowych, jak i plików binarnych, a także będą obejmowały nieco bardziej zaawansowane tryby przetwarzania plików, które mogą być dla Ciebie przydatne podczas pisania własnych programów.

Krotki

Ostatnim obiektem kolekcji w naszym przeglądzie typów podstawowych Pythona jest krotka (ang. *tuple*). Krotki tworzą proste grupy obiektów. Działają dokładnie tak jak listy, jednak nie mogą być modyfikowane w miejscu (są niemutowalne) i zazwyczaj są zapisywane jako seria elementów w zwykłych nawiasach okrągłych, a nie w nawiasach kwadratowych. Choć nie obsługują tak wielu metod, dzielą większość właściwości z listami. Poniżej znajduje się krótkie podsumowanie właściwości krotek.

Krotki są uporządkowanymi kolekcjami dowolnych obiektów

Tak jak łańcuchy znaków i listy, krotki są uporządkowanymi kolekcjami obiektów — zachowują zatem kolejność elementów od lewej do prawej strony. Tak jak w przypadku list, również w krotkach możemy osadzać dowolne typy obiektów.

Dostęp do krotek odbywa się po wartości przesunięcia

Podobnie jak w przypadku łańcuchów znaków i list, dostęp do elementów krotki odbywa się za pomocą wartości przesunięcia (a nie według klucza). Krotki obsługują wszystkie operacje oparte na wartościach przesunięcia, w tym indeksowanie i wycinki.

Krotki należą do kategorii sekwencji niemutowalnych

Tak jak łańcuchy znaków oraz listy, krotki są sekwencjami — obsługują wiele z operacji odnoszących się do tych dwóch typów obiektów. Tak samo jak łańcuchy znaków, krotki są jednak niemutowalne, stąd nie obsługują żadnych operacji modyfikujących w miejscu, które mają zastosowanie do list.

Krotki mają stałą długość, są heterogeniczne i można je dowolnie zagnieżdżać

Ponieważ krotki są niemutowalne, nie można zmieniać ich rozmiaru bez tworzenia kopii. Z drugiej strony, krotki mogą przechowywać dowolne typy obiektów, w tym inne obiekty złożone (na przykład listy, słowniki i inne krotki) i mogą być praktycznie dowolnie zagnieżdżane.

Krotki są tablicami referencji do obiektów

Podobnie jak listy, krotki najlepiej jest sobie wyobrazić jako tablice referencji do obiektów. Krotki przechowują punkty dostępu do innych obiektów (referencje), a indeksowanie krotek jest dość szybkie.

W tabeli 9.1 przedstawiono najczęściej wykonywane operacje na krotkach. Krotka zapisywana jest jako seria obiektów (z technicznego punktu widzenia: wyrażeń generujących obiekty) rozdzielonych przecinkami i zazwyczaj umieszczonych w zwykłych nawiasach okrągłych. Pusta krotka to po prostu para nawiasów bez zawartości.

Tabela 9.1. Popularne literały oraz operacje na krotkach

Operacja	Interpretacja
()	Pusta krotka
T = (0,)	Krotka jednoelementowa (nie jest wyrażeniem)
T = (0, 'Ni', 1.2, 3)	Krotka czteroelementowa
T = 0, 'Ni', 1.2, 3	Inna krotka czteroelementowa (ta sama co wyżej)
T = ('Adam', ('dev', 'mgr'))	Zagnieżdżone krotki

<code>T = tuple('mielonka')</code>	Krotka elementów w obiekcie iterowalnym
<code>T[i]</code> <code>T[i][j]</code> <code>T[i:j]</code> <code>len(T)</code>	Indeks, indeks indeksu, wycinek, długość
<code>T1 + T2</code> <code>T2 * 3</code>	Konkatenacja, powtórzenie
<code>for x in T: print(x)</code> <code>'mielonka' in T</code> <code>[x ** 2 for x in T]</code>	Iteracja, przynależność
<code>T.index('Ni')</code> <code>T.count('Ni')</code>	Metody z wersji 2.6, 2.7 i 3.x — wyszukiwanie, zliczanie
<code>namedtuple('Emp', ['name', 'jobs'])</code>	Krotka z nazwanymi polami (krotka typu named tuple)

Krotki w akcji

Jak zawsze, powinieneś teraz uruchomić sesję interaktywną, aby na własne oczy przekonać się, jak działają krotki. W tabeli 9.1 warto zwrócić uwagę na to, że krotki nie mają wszystkich metod posiadanych przez listy (zatem na przykład wywołanie metody `append` tutaj nie zadziała). Obsługują jednak zwykłe operacje na sekwencjach, które widzieliśmy już przy okazji omawiania zarówno łańcuchów znaków, jak i list.

```
>>> (1, 2) + (3, 4)                                # Konkatenacja
(1, 2, 3, 4)
>>> (1, 2) * 4                                    # Powtórzenie
(1, 2, 1, 2, 1, 2, 1, 2)
>>> T = (1, 2, 3, 4)                             # Indeksowanie, wycinek
>>> T[0], T[1:3]
(1, (2, 3))
```

Właściwości składni krotek — przecinki i nawiasy

Drugi i czwarty wpis z tabeli 9.1 zasługują na słowo wyjaśnienia. Ponieważ nawiasy mogą również zawierać wyrażenia (zobacz rozdział 5.), musimy jakoś przekazać Pythonowi, że pojedynczy obiekt w nawiasach jest obiektem krotki, a nie prostym wyrażeniem. Jeżeli naprawdę chcemy utworzyć krotkę jednoelementową, wystarczy w nawiasach po tym pojedynczym elemencie dopisać przecinek.

```
>>> x = (40)                                       # Liczba całkowita!
>>> x
40
>>> y = (40,)                                     # Krotka zawierająca pojedynczą
liczbę całkowitą
```

```
>>> y
```

```
(40,)
```

W tym specjalnym przypadku Python pozwala również na pominięcie nawiasu otwierającego i zamykającego dla krotki w kontekstach, w których nie jest to niejednoznaczne pod względem składniowym. Na przykład w czwartym wierszu tabeli 9.1 po prostu wymieniamy cztery elementy rozdzielone przecinkami. W kontekście instrukcji przypisania Python rozpozna taki zapis jako krotkę, nawet jeżeli nie ma ona nawiasów.

Niektórzy zalecają, by krotki zawsze umieszczać w nawiasach. Inni mówią, żeby w krotkach nigdy nie używać nawiasów (a jeszcze inni żyją własnym życiem i nie narzucają ludziom, co mają robić ze swoimi krotkami!). Najczęściej spotykane miejsca, w których nawiasy są *obowiązkowo wymagane*, to:

- *ze względu na nawiasy* — wywołania funkcji, gdzie przekazujemy krotki jako literały lub kiedy zagnieźdzamy krotki w bardziej złożonych wyrażeniach,
- *ze względu na przecinki* — osadzanie krotek w postaci literałów w większych strukturach danych, takich jak lista lub słownik, lub wyświetlanie ich za pomocą instrukcji `print` z Pythona 2.x.

W innych kontekstach umieszczanie krotek w nawiasach jest opcjonalne. Początkującym użytkownikom łatwiej będzie chyba używać nawiasów, niż zastanawiać się nad tym, kiedy są one opcjonalne. Wielu programistów (ze mną na czele) uważa również, że nawiasy zwiększą czytelność skryptów, bo dzięki nim krotki są lepiej widoczne i bardziej oczywiste^[1].

Konwersje, metody oraz niemutowalność

Poza różnicami w składni literałów, operacje na krotkach (trzy środkowe wiersze z tabeli 9.1) są identyczne z operacjami na łańcuchach znaków i listach. Jedyna różnica, jaką warto podkreślić, polega na tym, że operacje `+`, `*` oraz wycinki po zastosowaniu do krotek zwracają *nowe krotki*. Krotki nie posiadają również tych samych metod, z jakimi spotkaliśmy się przy okazji omawiania łańcuchów znaków, list oraz słowników. Kiedy chcemy na przykład posortować krotkę, zazwyczaj musimy ją najpierw przekonwertować na listę w celu uzyskania dostępu do wywołania metody sortującej i zmiany obiektu na mutowalny. Zamiast tego możemy także skorzystać z nowszej funkcji wbudowanej `sorted`, która przyjmuje dowolny obiekt sekwencji (oraz inne obiekty *iterowalne* — jest to określenie wprowadzone w rozdziale 4., którego będziemy używać w bardziej formalny sposób w następnej części tej książki):

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)                                     # Sporządzenie listy z elementów
krotki
>>> tmp.sort()                                       # Sortowanie listy
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)                                   # Sporządzenie krotki z
elementów listy
>>> T
('aa', 'bb', 'cc', 'dd')
>>> sorted(T)                                       # Można także użyć funkcji
wbudowanej sorted
['aa', 'bb', 'cc', 'dd']
```

W powyższym kodzie wbudowane funkcje `list` oraz `tuple` wykorzystano do przekształcenia obiektu na listę, a następnie konwersji tej listy z powrotem na krotkę. Oba wywołania tworzą nowe obiekty, jednak ich rezultat przypomina konwersję.

Do konwersji krotek można również wykorzystać listy składane. Poniższy kod tworzy z krotki listę, dodając przy okazji liczbę 20 do każdego elementu.

```
>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]
```

Listy składane są tak naprawdę operacjami na *sekwencjach* — zawsze tworzą nowe listy, jednak można je również wykorzystać do iteracji po dowolnych obiektach sekwencji, w tym krotkach, łańcuchach znaków i innych listach. Jak zobaczymy nieco później, działają nawet na niektórych elementach niebędących fizycznie przechowywanymi sekwencjami — wystarczą dowolne obiekty, po których można *iterować*, w tym pliki, które automatycznie są ładowane wiersz po wierszu. Biorąc to pod uwagę, powinniśmy je raczej nazywać narzędziami *iteracyjnymi*.

Choć krotki nie obsługują tych samych metod, co listy i łańcuchy znaków, w Pythonie 2.6 oraz 3.0 mają dwie własne metody, `index` oraz `count`, które działają tak samo jak w przypadku list, jednak zdefiniowane są dla obiektów krotek.

```
>>> T = (1, 2, 3, 2, 4, 2)                                # Metody krotek w Pythonie 2.6 oraz 3.0
      i nowszych wersjach

>>> T.index(2)                                         # Offset pierwszego wystąpienia
elementu o wartości 2

1

>>> T.index(2, 2)                                     # Offset wystąpienia elementu o
wartości 2 po przesunięciu o 2 pozycje

3

>>> T.count(2)                                       # Ile jest elementów o wartości 2?

3
```

Przed Pythonem 2.6 oraz 3.0 krotki nie miały żadnych metod — była to stara konwencja Pythona dla typów niemutowalnych, która z powodu niepraktyczności została zarzucona wiele lat temu dla łańcuchów znaków, a ostatnio — dla liczb oraz krotek.

Warto również zauważyc, że reguła dotycząca *niemutowalności* krotki odnosi się jedynie do jej najwyższego poziomu, a nie do całej zawartości — na przykład lista znajdująca się wewnątrz krotki może być normalnie zmieniana.

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'mielonka'                                    # Nie działa – nie da się zmienić
krotki

TypeError: object doesn't support item assignment

>>> T[1][0] = 'mielonka'                               # Działa – można modyfikować mutowalne
elementy krotki

>>> T
(1, ['mielonka', 3], 4)
```

W większości programów taka niezmiennosć na najwyższym poziomie jest zupełnie wystarczająca w zastosowaniach, w których najczęściej pojawiają się krotki. Zagadnienie to przenosi nas również do kolejnego punktu.

Dlaczego istnieją listy i krotki

To chyba pierwsze pytanie, jakie pojawia się przy omawianiu krotek — po co nam one, skoro mamy listy? Uzasadnienie jest częściowo historyczne — twórca Pythona jest z wykształcenia matematykiem i wspomina się, że krotki wyobrażały sobie jako proste powiązanie obiektów, natomiast listy — jako struktury danych zmieniające się z czasem. Tak naprawdę samo słowo „krotka” pochodzi z matematyki, podobnie jak jej częste zastosowanie dla wiersza w tabeli relacyjnej bazy danych.

Najtrawniejszą odpowiedzią na to pytanie wydaje się jednak ta, że niemutowalność krotek zapewnia pewien stopień *integralności* — możemy być pewni, że krotka nie zostanie zmodyfikowana przez inną referencję umieszoną gdzieś w programie; w przypadku list takiej gwarancji nie ma. Krotki i inne obiekty niemutowalne pełnią zatem rolę podobną do stałych z innych języków programowania, choć w Pythonie „stałość” powiązana jest z obiektami, a nie zmiennymi.

Krotki można również wykorzystywać w miejscach, w których nie można użyć list — na przykład jako klucze słowników (zobacz przykład rzadkiej macierzy w rozdziale 8.). Niektóre wbudowane operacje mogą również wymagać lub sugerować użycie krotek zamiast list (np. wartości podstawień w wyrażeniach formatujących), chociaż takie operacje często były uogólniane w celu zapewnienia im większej elastyczności. Generalnie listy są narzędziem wybieranym dla uporządkowanych kolekcji obiektów, które mogą się zmieniać, a krotki lepiej sprawdzają się w przypadkach stałych powiązań.

Repetitorium: rekordy — krotki nazwane

W rzeczywistości wybór typów danych jest jeszcze bogatszy, niż sugerowała to poprzednia sekcja — dzisiejsi programiści Pythona mogą wybierać spośród asortymentu zarówno wbudowanych typów podstawowych, jak i opartych na nich typów rozszerzonych. W poprzednim rozdziale, w ramce „Słowniki kontra listy”, pokazywaliśmy, jak za pomocą zarówno listy, jak i słownika możemy reprezentować informacje o strukturze rekordów, i zauważaliśmy, że zaletą słowników są klucze mnemoniczne spełniające rolę etykiet danych. Warto jednak zauważać, że dopóty, dopóki nie wymagamy mutowalności, krotki mogą pełnić podobną rolę — w takim scenariuszu odpowiednikami pól rekordów są elementy na kolejnych pozycjach krotki:

```
>>> adam = ('Adam', 40.5, ['dev', 'mgr']) # Rekord w postaci krotki  
>>> adam  
('Adam', 40.5, ['dev', 'mgr'])  
>>> adam[0], adam[2] # Dostęp do pól według pozycji w krotce  
('Adam', ['dev', 'mgr'])
```

Jednak w przypadku list numery pól w krotkach zwykle zawierają mniej informacji niż nazwy kluczowe w słowniku. Oto ten sam rekord przekodowany jako słownik z nazwanymi polami:

```
>>> adam = dict(name='Adam', age=40.5, jobs=['dev', 'mgr']) # Rekord w postaci słownika  
>>> adam
```

```

{'jobs': ['dev', 'mgr'], 'name': 'Adam', 'age': 40.5}
>>> adam['name'], adam['jobs']                                # Dostęp
według kluczy
('Adam', ['dev', 'mgr'])

```

W razie potrzeby możemy przekonwertować części słownika na krotkę:

```

>>> tuple(adam.values())                                     # Zamiana
wartości na krotkę
(['dev', 'mgr'], 'Adam', 40.5)
>>> list(adam.items())                                       # Zamiana
elementów na listę krotek
[('jobs', ['dev', 'mgr']), ('name', 'Adam'), ('age', 40.5)]

```

Jednak przy odrobinie dodatkowej pracy możemy zaimplementować obiekty, które oferują zarówno pozycyjny, jak i nazwany dostęp do pól rekordów. Na przykład typ `namedtuple`, dostępny w standardowym module `collections`, o którym wspominaliśmy w rozdziale 8., to rozszerzenie, które dodaje krotkom funkcjonalność umożliwiającą dostęp do ich wartości zarówno według *pozycji* elementu, jak i *nazwy* atrybutu oraz zapewnia możliwość przekonwertowania takiej hybrydowej krotki na komponent zbliżony do słownika, pozwalający na dostęp do jego wartości według kluczy. Nazwy atrybutów pochodzą z klas i nie są dokładnie kluczami słownikowymi, ale są podobnie mnemoniczne:

```

>>> from collections import namedtuple                      # Import typu
rozszerzonego
>>> Rec = namedtuple('Rec', ['name', 'age', 'jobs'])       # Utworzenie
instancji klasy
>>> adam = Rec('Adam', age=40.5, jobs=['dev', 'mgr'])    # Rekord w postaci
nazwanej krotki
>>> adam
Rec(name='Adam', age=40.5, jobs=['dev', 'mgr'])
>>> adam[0], adam[2]                                         # Dostęp według
pozycji elementów
('Adam', ['dev', 'mgr'])
>>> adam.name, adam.jobs                                    # Dostęp według
atrybutów
('Adam', ['dev', 'mgr'])

```

W razie potrzeby możemy dokonać konwersji takiego rekordu na słownik, zyskując tym samym obsługę dostępu opartą na kluczach:

```

>>> o = adam._asdict()                                      # Rekord w postaci
zbliżonej do słownika
>>> o['name'], o['jobs']                                    # Dostęp według
kluczy
('Adam', ['dev', 'mgr'])
>>> o
OrderedDict([('name', 'Adam'), ('age', 40.5), ('jobs', ['dev', 'mgr'])])

```

Jak widać, nazwane krotki są hybrydami krotki, klasy oraz słownika i są przykładem klasycznego *kompromisu*. W zamian za dodatkową funkcjonalność wymagają napisania dodatkowego kodu (dwa wiersze startowe w poprzednich przykładach, które importują typ i tworzą instancję klasy), który może mieć dodatkowy wpływ na wydajność. Inaczej mówiąc, nazwane krotki to nowa klasa, która rozszerza podstawowy typ krotki, dodając metodę `property` dla każdego nazwanego pola, odwzorowującego nazwę pola na jego pozycję — jest to technika oparta na bardziej zaawansowanych zagadnieniach, które będziemy omawiać w części VIII. Nazwane krotki używają ciągów formatujących zamiast narzędzi adnotacji klas, takich jak dekoratory czy metaklasy. Mimo to są dobrym przykładem niestandardowych typów danych, które w sytuacji, gdy potrzebne jest nowe narzędzie, możemy tworzyć na bazie typów wbudowanych, takich jak krotki.

Krotki nazwane są dostępne w Pythonie 3.x, 2.7, 2.6 (gdzie funkcja `_asdict` zwraca prawdziwy słownik), a częściowo i we wcześniejszych wersjach, chociaż opierają się na funkcjach stosunkowo nowoczesnych według standardów Pythona. Są to również rozszerzenia, a nie podstawowe typy — znajdują się w standardowej bibliotece i należą do tej samej kategorii, co typy `Fraction` i `Decimal`, omawiane w rozdziale 5., zatem więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji Pythona.

Warto pamiętać, że zarówno krotki, jak i krotki nazwane obsługują mechanizm przypisania rozpakowującego (ang. *unpacking tuple assignment*), o którym będziemy mówić w rozdziale 13., a także *konteksty iteracyjne* (ang. *iteration contexts*), które zbadamy w rozdziale 14. i rozdziale 20. W przykładzie poniżej zwróć uwagę na początkowe wartości pozycyjne — nazwane krotki akceptują je według nazwy, pozycji lub obu tych elementów jednocześnie:

```
>>> adam = Rec('Adam', 40.5, ['dev', 'mgr'])          # Zarówno dla zwykłych,  
jak i nazwanych krotek  
  
>>> name, age, jobs = adam                         # Przypisanie krotek  
(zobacz rozdział 11.)  
  
>>> name, jobs  
('Adam', ['dev', 'mgr'])  
  
>>> for x in adam: print(x)                         # Kontekst iteracyjny  
(zobacz rozdziały 14. i 20.)  
  
...wyświetla Adam, 40.5, ['dev', 'mgr']...
```

Przypisywanie rozpakowujące nie dotyczy w pełni słowników, ponieważ brakuje tutaj pobierania i konwertowania kluczy i wartości oraz zakładania lub narzucania kolejności elementów (słowniki nie są sekwencjami), a iteracje przechodzą przez kolejne klucze, a nie wartości. W przykładzie poniżej zwróć uwagę na literalną formę słownika; jest to alternatywa dla konstruktora `dict`:

```
>>> adam = {'name': 'Adam', 'age': 40.5, 'jobs': ['dev', 'mgr']}
```

Przykład poniżej pokazuje, jak można uzyskać odpowiednik słownika (ale kolejność elementów może się różnić)

```
>>> job, name, age = adam.values()  
# Odpowiednik słownika (ale kolejność  
elementów może się różnić)  
('Adam', ['dev', 'mgr'])
```

Przykład poniżej pokazuje, jak można iterować po kluczach, indeksując wartości

```
>>> for x in adam: print(adam[x])      # Iterujemy po kluczach, indeksujemy  
wartości  
  
...wyświetlamy wartości...  
  
>>> for x in bob.values(): print(x)    # Iterujemy po widoku wartości  
  
...wyświetlamy wartości...
```

W rozdziale 27. jeszcze raz powrócimy do reprezentacji rekordów podczas omawiania zagadnień związanych z porównywaniem za pomocą *klas zdefiniowanych przez użytkownika*; jak się przekonasz, klasy również pozwalają na opisywanie pól nazwami, ale mogą także dostarczać *logikę* programu niezbędną do przetwarzania danych rekordu w tym samym pakiecie.

Pliki

Większość użytkowników spotkała się już z pojęciem plików, czyli nazwanych kontenerów przechowujących dane w komputerze, którymi zarządza system operacyjny. Ostatni z podstawowych, wbudowanych typów obiektów, o których mówimy podczas naszej prezentacji najważniejszych typów obiektów, zapewnia dostęp do plików z poziomu programów napisanych w języku Python.

Krótko mówiąc, wbudowana funkcja `open` tworzy obiekt pliku, który służy jako łącze do pliku znajdującego się w komputerze. Po wywołaniu tej funkcji możesz przesyłać ciągi danych do i z powiązanego pliku zewnętrznego, wywołując odpowiednie metody obiektu reprezentującego taki plik.

W porównaniu z innymi typami obiektów, z jakimi się dotychczas spotykaliśmy, obiekty reprezentujące pliki są w pewien sposób niezwykłe. Są uważane za typ podstawowy, ponieważ tworzymy je za pomocą wbudowanej funkcji, ale nie są liczbami, sekwencjami czy odwzorowaniami, ani też nie reagują na operatory wyrażeń; zamiast tego udostępniają jedynie metody służące do wykonywania typowych zadań związanych z przetwarzaniem plików. Większość metod plików zajmuje się pobieraniem danych wejściowych z plików zewnętrznych i zapisywaniem do nich danych wyjściowych. Inne metody pozwalają na odszukiwanie określonej pozycji w pliku czy opróżnianie bufora wyjściowego. W tabeli 9.2 przedstawiono popularne operacje na plikach.

Tabela 9.2. Popularne operacje na plikach

Operacja	Interpretacja
<code>output = open(r'C:\spam', 'w')</code>	Tworzy plik do zapisu ('w' z ang. <i>write</i> — zapis)
<code>input = open('data', 'r')</code>	Otwiera plik do odczytu ('r' z ang. <i>read</i> — odczyt)
<code>input = open('data')</code>	To samo co wyżej ('r' jest trybem domyślnym)
<code>aString = input.read()</code>	Załadowanie całej zawartości pliku do jednego łańcucha znaków
<code>aString = input.read(N)</code>	Załadowanie kolejnych <i>N</i> bajtów (jednego lub więcej) do łańcucha znaków
<code>aString = input.readline()</code>	Załadowanie kolejnego wiersza (wraz ze znakami nowego wiersza \n) do łańcucha znaków
<code>aList = input.readlines()</code>	Załadowanie całej zawartości pliku do listy łańcuchów znaków (wraz ze znakami \n)
<code>output.write(aString)</code>	Zapisanie ciągu znaków (lub bajtów) do pliku
<code>output.writelines(aList)</code>	Zapisanie wszystkich łańcuchów znaków z listy do pliku

<code>output.close()</code>	Ręczne zamknięcie pliku (kiedy kończymy pracę z plikiem)
<code>output.flush()</code>	Zapisanie zawartości bufora wyjściowego na dysku bez zamazywania pliku
<code>anyFile.seek(<i>N</i>)</code>	Zmiana pozycji w pliku na offset <i>N</i> dla następnej operacji
<code>for wiersz in open('data'): użycie wiersza</code>	Iteratory plików — wczytywanie zawartości pliku wiersz po wierszu
<code>open('f.txt', encoding='latin-1')</code>	Pliki Unicode w Pythonie 3.x (ciągi typu str)
<code>open('f.bin', 'rb')</code>	Pliki bajtowe w Pythonie 3.x (ciągi typu bytes)
<code>codecs.open('f.txt', encoding='utf8')</code>	Pliki Unicode w Pythonie 2.x (ciągi typu unicode)
<code>open('f.bin', 'rb')</code>	Pliki bajtowe w Pythonie 2.x (ciągi typu str)

Otwieranie plików

Aby otworzyć plik, program wywołuje wbudowaną funkcję `open`, przekazując jej nazwę pliku zewnętrznego oraz *tryb* przetwarzania. Wynikiem działania funkcji jest obiekt pliku, który posiada odpowiednie metody przesyłania danych:

```
afile = open(nazwa_pliku, tryb_przetwarzania)
afile.metoda()
```

Pierwszy argument wywołania funkcji `open` to *nazwa_pliku*, która może być poprzedzona specyficzną dla platformy ścieżką katalogu. Jeżeli ścieżka nie zostanie podana, funkcja domyślnie przyjmuje, że plik znajduje się w bieżącym katalogu roboczym (np. w katalogu, z którego został uruchomiony skrypt). Jak zobaczymy w rozdziale 37., *nazwa_pliku* może również zawierać znaki Unicode inne niż ASCII, które Python automatycznie tłumaczy z domyślnego kodowania platformy bazowej i na odwrót, lub może być przekazana jako wstępnie zakodowany ciąg bajtów.

Drugi argument wywołania funkcji `open`, reprezentujący tryb przetwarzania, to zazwyczaj łańcuch znaków '*r*' dla odczytywania pliku (wartość domyślna), '*w*' dla utworzenia pliku i otwarcia go do zapisu lub '*a*' dla dodawania tekstu na końcu pliku (na przykład dla dołączania nowych wpisów do pliku dziennika). Argument trybu przetwarzania może posiadać również dodatkowe opcje:

- Dodanie znaku *b* do argumentu trybu pozwala na odczytywanie i zapisywanie *danych binarnych* (wyłączane jest interpretowanie znaków końca wiersza oraz kodowanie Unicode z Pythona 3.x).
- Dodanie znaku *+* oznacza, że plik otwierany jest *zarówno* do odczytu, jak i zapisu (możemy zatem odczytywać i zapisywać dane w tym samym pliku, często w połączeniu z operacjami wyszukiwania służącymi do zmiany pozycji w pliku).

Oba argumenty funkcji `open` muszą być łańcuchami znaków Pythona, natomiast opcjonalny trzeci argument można wykorzystać do kontrolowania *bufora wyjściowego* — np. przekazanie zera oznacza, że dane wyjściowe nie będą buforowane (będą przenoszone do pliku zewnętrznego natychmiast po wywołaniu metody zapisu), a dla specjalnych typów plików

można podawać dodatkowe argumenty (np. rodzaj kodowania plików tekstowych Unicode w Pythonie 3.x).

W tym rozdziale omówimy podstawowe zagadnienia związane z pracą z plikami i przedstawimy kilka prostych przykładów, natomiast nie będziemy się bardziej zagłębiać w opcje trybu przetwarzania plików. Więcej informacji na ten temat można, jak zwykle, znaleźć w dokumentacji Pythona.

Wykorzystywanie plików

Po utworzeniu pliku za pomocą funkcji `open` możemy wywoływać jego metody w celu odczytania danych z pliku lub zapisania danych do pliku. W każdym z tych przypadków tekst pliku przybiera w programach w Pythonie postać łańcuchów znaków. Odczytanie pliku powoduje zwrócenie jego zawartości w postaci tekstu osadzonego w łańcuchach znaków i analogiczne łańcuchy tekstu przekazujemy do metod zapisujących dane do plików. Metody odczytujące i zapisujące dane mają wiele odmian; w tabeli 9.2 zaprezentowano te najbardziej popularne. Poniżej zamieszczamy kilka najważniejszych uwag dotyczących używania plików.

Do odczytywania wierszy najlepiej nadają się iteratory plików

Choć wymienione w tabeli metody służące do odczytu i zapisu są dość popularne, warto pamiętać, że chyba najlepszym sposobem na odczytywanie wierszy z pliku tekstowego wcale nie jest załadowanie pliku. Jak zobaczymy w rozdziale 14., pliki posiadają *iterator* automatycznie odczytujący po jednym wierszu na raz w pętli `for`, liście składanej czy innym kontekście iteracyjnym.

Zawartość pliku to łańcuchy znaków, a nie obiekty

W tabeli 9.2 warto zwrócić uwagę na to, że dane wczytane z pliku zawsze trafiają do skryptu w postaci łańcucha znaków. Jeżeli zatem chcesz korzystać z innego typu obiektu Pythona, będziesz musiał najpierw dokonać konwersji. I podobnie — w przeciwnieństwie do sytuacji z operacją `print` — Python przy zapisie danych do pliku nie dodaje żadnego formatowania i nie przekształca obiektów na łańcuchy znaków automatycznie; do pliku należy przesłać uprzednio sformatowany łańcuch znaków. Z tego powodu poznane wcześniej narzędzia (jak `int`, `float`, `str` i wyrażenia formatujące łańcuchy znaków) konwertujące obiekty na łańcuchy znaków (i odwrotnie) przydadzą się także przy pracy z plikami.

Python w swoich bibliotekach posiada również zaawansowane narzędzia do obsługi przechowywania obiektów (moduł `pickle`), do obsługi spakowanych danych binarnych w plikach (moduł `struct`) oraz do przetwarzania specjalnych typów zawartości, takich jak tekst w formacie JSON, XML czy CSV. Przykłady ich zastosowania znajdziesz w dalszej części tego rozdziału i kolejnych rozdziałach książki, a więcej szczegółowych informacji zawiera dokumentacja Pythona.

Pliki są buforowane i można je przeszukiwać

Domyślnie pliki wyjściowe są zawsze buforowane, co oznacza, że tekst, który piszemy, nie zawsze jest natychmiast przenoszony z pamięci na dysk — dopiero zamknięcie pliku lub wywołanie jego metody `flush` wymusza transfer danych na dysk. Buforowanie można wyłączyć za pomocą dodatkowych argumentów metody `open`, jednak może to mieć negatywny wpływ na wydajność. Python zapewnia bezpośredni, swobodny dostęp do zawartości plików poprzez określanie pozycji przesunięcia (offsetu) względem początku pliku — metoda `seek` pozwala skryptom na odczytywanie i zapisywanie danych w określonych pozycjach.

Wywołanie metody `close` jest zazwyczaj opcjonalne

Wywołanie metody `close` kończy połączenie z plikiem zewnętrznym, uwalnia powiązane z nim zasoby systemowe i opróżnia buforowane dane wyjściowe na dysk, jeżeli były jeszcze w pamięci. Jak wspominaliśmy w rozdziale 6., przestrzeń pamięci obiektu jest automatycznie zwalniana, kiedy w programie nie ma już żadnych referencji do tego obiektu. Kiedy *obiekty plików* są zwalniane, Python automatycznie *zamyka* również same pliki, jeżeli nadal były otwarte (to samo ma miejsce, gdy zamykamy program). Oznacza to, że w programach napisanych w standardowym języku Python nie zawsze musisz ręcznie zamykać pliki, zwłaszcza w prostych, krótko działających skryptach czy w plikach tymczasowych używanych przez pojedynczy wiersz polecenia lub wyrażenie.

Z drugiej strony ręczne zamykanie połączeń nikomu nie szkodzi i może być dobrym nawykiem, szczególnie w systemach, które działają przez dłuższy czas. Ścisłe mówiąc, funkcja automatycznego zamykania plików nie jest częścią definicji języka — jej działanie w kolejnych wersjach Pythona może się zmieniać, może nie działać tak, jak się tego spodziewasz, w interaktywnej sesji powłoki i może nie działać tak samo w innych implementacjach Pythona, w których mechanizmy odzyskiwania zasobów mogą działać w nieco inny sposób niż w standardowym CPythonie. W rzeczywistości, gdy w pętlach otwieranych jest wiele plików, implementacje inne niż CPython mogą wymagać wywoływania metody `close` w celu zwolnienia zasobów systemowych, zanim automatyczny proces odzyskiwania zasobów będzie w stanie uwolnić te obiekty. Co więcej, jawne wywołania metody `close` mogą czasami być konieczne w celu opróżnienia buforowanych danych wyjściowych obiektów plików, których zasoby nie zostały jeszcze odzyskane automatycznie. Aby poznać alternatywny sposób zagwarantowania automatycznego zamykania plików, powinieneś zatrzymać się na chwilę i omówić mechanizmy *kontekstu* obiektu pliku, które znajdziesz w dalszej części tego rozdziału. W Pythonie 2.6, 2.7 i 3.x menedżery kontekstu plików są używane w połączeniu z wyrażeniami `with/as`.

Pliki w akcji

Przyjrzyjmy się teraz prostemu przykładowi demonstrującemu podstawy przetwarzania plików. Poniższy kod rozpoczyna się od otwarcia nowego pliku tekstowego do zapisu, później zapisuje do tego pliku dwa wiersze (łańcuchy znaków zakończone znacznikiem nowego wiersza `\n`), a następnie plik został zamknięty. Potem kod ponownie otwiera ten sam plik, tym razem w trybie do odczytu, i wczytuje umieszczone w nim wiersze — za pomocą metody `readline`, każdorazowo po jednym. Warto zwrócić uwagę na to, że trzecie wywołanie metody `readline` zwraca pusty łańcuch znaków. W ten sposób metody plików Pythona przekazują nam, że osiągnęliśmy koniec pliku (puste wiersze z pliku zwracane są jako łańcuchy znaków zawierające znak nowego wiersza, a nie jako puste łańcuchy znaków). Poniżej zamieszczamy kod przykładu.

```
>>> myfile = open('myfile.txt', 'w')          # Otwarcie do zapisu tekstu
(tworzy pusty plik)
>>> myfile.write('witaj, pliku tekstowy\n')    # Zapisanie wiersza tekstu
22
>>> myfile.write('żegnaj, pliku tekstowy\n')
23
>>> myfile.close()                          # Zrzucenie bufora wyjściowego
na dysk
>>> myfile = open('myfile.txt')            # Otwarcie do odczytu tekstu -
'r' jest domyślne
>>> myfile.readline()                     # Wczytanie wierszy z powrotem
'witaj, pliku tekstowy\n'
```

```

>>> myfile.readline()
'żegnaj, pliku tekstowy\n'
>>> myfile.readline()                                # Pusty łańcuch znaków – koniec
pliku
''
```

Warto zauważyc, że wywołania metody `write` pliku zwracają w Pythonie 3.x liczbę znaków. W wersji 2.x tak nie jest, dlatego w sesji interaktywnej nie zobaczymy tych liczb. Powyższy przykład zapisuje każdy z wierszy tekstu (wraz ze znacznikiem kończącym wiersz `\n`) w postaci łańcucha znaków. Metody zapisujące do plików nie dodają znaku końca wiersza za nas, dlatego musimy to zrobić sami, by w poprawny sposób zakończyć wiersz tekstu. Jeśli tego nie zrobimy, kolejny zapis rozszerzy aktualny wiersz pliku.

Jeżeli chcemy wyświetlić zawartość pliku ze zinterpretowanymi znakami końca wiersza, należy wczytać cały plik do jednego łańcucha znaków *naraz* za pomocą metody `read` obiektu pliku, a następnie wyświetlić go.

```

>>> open('myfile.txt').read()                      # Wczytanie wszystkiego naraz do
                                                 łańcucha znaków
'witaj, pliku tekstowy\nżegnaj, pliku tekstowy\n'
>>> print(open('myfile.txt').read())               # Sposób wyświetlania przyjazny
                                                 dla użytkownika
witaj, pliku tekstowy
żegnaj, pliku tekstowy

Jeżeli natomiast chcemy przejrzeć plik tekstowy wiersz po wierszu, najlepszą opcją jest często skorzystanie z iteratorów plików.
```

```

>>> for line in open('myfile'):                   # Użycie iteratorów plików, a nie
                                                 wczytywania
...     print(line, end='')

...
witaj, pliku tekstowy
żegnaj, pliku tekstowy
```

Przy takim kodzie tymczasowy obiekt pliku utworzony za pomocą metody `open` automatycznie wczyta i zwróci po jednym wierszu z każdą iteracją pętli. Forma ta jest zazwyczaj najłatwiejsza do utworzenia w kodzie, charakteryzuje się niezłym zużyciem pamięci i może być szybsza od części innych rozwiązań (co oczywiście uzależnione jest od innych czynników). Ponieważ nie dotarliśmy jeszcze do instrukcji ani iteratorów, na pełne wyjaśnienie powyższego kodu będziemy musieli poczekać do rozdziału 14.



Użytkownicy systemu Windows: Jak wspominaliśmy w rozdziale 7., w systemie Windows funkcja `open` w ścieżkach do plików akceptuje również uniksowe prawe ukośniki zamiast tradycyjnych lewych ukośników, więc wszystkie przedstawione niżej przykłady wywołania tej funkcji będą działać poprawnie (nieprzetworzone ścieżki z lewymi ukośnikami, prawymi ukośnikami lub podwójnymi lewymi ukośnikami):

```

>>> open(r'C:\Python33\Lib\pdb.py').readline()
'#! /usr/bin/env python3\n'
```

```
>>> open('C:/Python33/Lib/pdb.py').readline()
#!/usr/bin/env python3\n'
>>> open('C:\\Python33\\Lib\\pdb.py').readline()
#!/usr/bin/env python3\n'
```

Możliwość używania nieprzetworzonych ciągów znaków, pokazana w pierwszym poleceniu, może być przydatna do wyłączania interpretowania przypadkowych znaków ucieczki, gdy nie można kontrolować zawartości ciągów znaków reprezentujących ścieżki oraz w innych kontekstach.

Pliki tekstowe i binarne – krótka historia

Warto zauważyć, że przykłady z poprzedniego podrozdziału wykorzystują pliki tekstowe. Zarówno w Pythonie 3.x, jak i 2.x typ pliku określa się za pomocą drugiego argumentu metody `open`, czyli łańcucha reprezentującego tryb przetwarzania pliku. Dołączenie do tego argumentu znaku „`b`” oznacza tryb binarny. Python zawsze obsługiwał zarówno pliki tekstowe, jak i binarne, jednak w wersjach 3.x rozróżnienie między tymi rodzajami plików jest znacznie mocniejsze:

- *Pliki tekstowe* reprezentują zawartość w postaci normalnych łańcuchów znaków `str`, wykonując automatycznie kodowanie i dekodowanie Unicode oraz domyślnie interpretując znaki końca wierszy.
- *Pliki binarne* reprezentują zawartość w postaci specjalnego typu `bytes` i pozwalają programom na dostęp do niezmienionej zawartości plików.

W przeciwieństwie do tego pliki tekstowe w Pythonie 2.x obsługują zarówno tekst 8-bitowy, jak i dane binarne; specjalny typ łańcucha znaków oraz interfejs plików (łańcuchy `unicode` oraz metoda `codecs.open`) obsługują tekst Unicode. Różnice w Pythonie 3.x wynikają z faktu, że zwykły tekst oraz Unicode zostały połączone w jeden typ łańcucha znaków — co ma sens, biorąc pod uwagę, że cały tekst, w tym ASCII oraz inne 8-bitowe systemy kodowania, to w rzeczywistości Unicode.

Ponieważ spora część programistów ma do czynienia jedynie z tekstem ASCII, wystarczające będą dla nich podstawowy interfejs plików tekstowych, wykorzystany w poprzednim przykładzie, oraz normalne łańcuchy znaków. W Pythonie 3.x wszystkie łańcuchy znaków są Unicode, czego użytkownicy ASCII zazwyczaj nawet nie zauważą. Tak naprawdę pliki i łańcuchy znaków działają w wersjach 3.x oraz 2.x tak samo, jeżeli zakres naszego skryptu ograniczony jest do takich prostych form tekstu.

Jeżeli musimy obsłużyć aplikacje wielojęzyczne lub przetwarzające dane bajtowe, to różnice, jakie występują w Pythonie 3.x, będą miały poważny wpływ na nasz kod (zazwyczaj na lepsze). Mówiąc ogólnie, dla plików binarnych musimy użyć typu `bytes`; zwykłe łańcuchy znaków `str` przeznaczone są dla plików tekstowych. Co więcej, ponieważ pliki tekstowe implementują kodowanie Unicode, nie można otworzyć pliku z danymi binarnymi w trybie tekstowym — dekodowanie jego zawartości na tekst Unicode z dużym prawdopodobieństwem się nie powiedzie.

Przyjrzyjmy się przykładowi. Po wczytaniu pliku z danymi *binarnymi* otrzymujemy obiekt typu `bytes` — sekwencję liczb całkowitych reprezentujących bezwzględne wartości bajtowe (które mogą, ale nie muszą, odpowiadać znakom), która wygląda i działa prawie dokładnie tak samo jak normalny łańcuch znaków. Przykład pokazany poniżej został przygotowany w Pythonie 3.x dla pliku binarnego:

```
>>> data = open('data.bin', 'rb').read()      # Otwarcie pliku binarnego: rb =
,,read binary"
```

```
>>> data                                # Łąćuch bytes przechowuje dane
binarne

b'\x00\x00\x00\x07mielonka\x00\x08'

>>> data[4:12]                            # Działa jak łańcuch znaków

b'mielonka'

>>> data[4:12][0]                         # Ale tak naprawdę to małe, 8-
bitowe liczby całkowite

109

>>> bin(data[4:12][0])                   # Funkcja bin(); Python 3.x/26+

'0b1101101'
```

Ponadto w przypadku plików binarnych znaki *konca wiersza* nie są interpretowane; w wersji 3.x pliki tekstowe domyślnie odwzorowują sekwencje znaków \n podczas zapisywania i odczytywania oraz domyślnie implementują kodowanie Unicode. Pliki binarne, takie jak pokazany w przykładzie, działają tak samo w Pythonie 2.x, ale ciągi bajtów są po prostu normalnymi ciągami i przy wyświetlaniu nie są poprzedzane żadnymi znakami b; pliki tekstowe do przetwarzania Unicode muszą używać modułu codecs.

Zgodnie z zapowiedzią na początku tego rozdziału w tym miejscu książki to już wszystko, co powiemy o tekstuach Unicode i plikach danych binarnych, ale to w zupełności wystarczy, aby zrozumieć przykłady omawiane w tym rozdziale. Ponieważ rozróżnienie tych rodzajów plików dla wielu programistów Pythona ma marginalne znaczenie, przypominamy, że krótki opis plików znajdziesz w rozdziale 4., i odkładamy omówienie pozostałych zagadnień związanych z plikami do rozdziału 37., a w dalszej części bieżącego rozdziału przejdziemy do zaprezentowania kilku typowych przykładów użycia plików.

Przechowywanie obiektów Pythona w plikach i przetwarzanie ich

Kolejny przykład zapisuje różne obiekty Pythona do pliku tekstuowego z kilkoma wierszami. Warto pamiętać, że kod ten musi przekształcić obiekty nałańcuchy znaków za pomocą odpowiednich narzędzi konwersji. Powtórzmy raz jeszcze: dane z plików są zawsze w skryptach *łańcuchami* znaków, natomiast metody zapisu nie formatują za nas obiektów nałańcuchy znaków. Z uwagi na brak miejsca pomijam odtąd wartości zliczania bajtów zwracane przez metodę `write`.

```
>>> X, Y, Z = 43, 44, 45 # Natywne obiekty Pythona muszą
być łańcuchami znaków,
>>> S = 'Mielonka' # by można je było umieścić w
plikach
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>>
>>> F = open('datafile.txt', 'w') # Utworzenie wyjściowego pliku
tekstowego
>>> F.write(S + '\n') # Zakończenie wierszy znakiem \n
```

```

>>> F.write('%s,%s,%s\n' % (X, Y, Z))      # Konwersja liczb na łańcuchy
znaków

>>> F.write(str(L) + '$' + str(D) + '\n')    # Konwersja i rozdzielenie
znakiem $

>>> F.close()

```

Po utworzeniu pliku możemy sprawdzić jego zawartość, otwierając go i wczytując do łańcucha znaków (w jednej operacji). Warto zwrócić uwagę, że w sesji interaktywnej zwracana jest dokładna zawartość bajtowa, natomiast operacja `print` interpretuje osadzone znaki końca wiersza, tak by wynik był bardziej przyjazny dla użytkownika.

```

>>> chars = open('datafile.txt').read()          # Wyświetlenie "surowego"
                                             # łańcucha znaków

>>> chars

"Mielonka\n43,44,45\n[1, 2, 3]$('a': 1, 'b': 2)\n"

>>> print(chars)                                # Wyświetlenie w sposób
                                             # przyjazny dla użytkownika

Mielonka
43,44,45
[1, 2, 3]$('a': 1, 'b': 2)

```

Teraz musimy skorzystać z narzędzi do konwersji w celu przekształcenia łańcuchów znaków z pliku tekstowego na prawdziwe obiekty Pythona. Ponieważ Python nigdy nie konwertuje łańcuchów znaków na liczby czy inne typy obiektów w sposób automatyczny, jest to konieczne, jeśli chcemy uzyskać dostęp do normalnych narzędzi obiektów, takich jak indeksowanie czy dodawanie.

```

>>> F = open('datafile.txt')                      # Ponowne otwarcie pliku

>>> line = F.readline()                          # Wczytanie jednego wiersza

>>> line

'Mielonka\n'

>>> line.rstrip()                               # Usunięcie znaku końca wiersza

'Mielonka'

```

W przypadku pierwszego wiersza do pozbycia się końcowego znaku nowego wiersza wykorzystaliśmy metodę `rstrip`. Wycinek `line[:-1]` również mógłby tutaj zadziałać, ale tylko wtedy, gdy bylibyśmy pewni, że wszystkie wiersze mają na końcu znak `\n` (czasami brakuje go w ostatnim wierszu).

Jak na razie udało nam się wczytać wiersz zawierający łańcuch znaków. Teraz pobierzemy kolejny wiersz, zawierający liczby, i dokonamy ekstrakcji obiektów z tego wiersza.

```

>>> line = F.readline()                          # Kolejny wiersz pliku

>>> line                                       # Tutaj jest on łańcuchem znaków

'43,44,45\n'

>>> parts = line.split(',')                   # Podział na przecinkach

>>> parts

```

```
[ '43', '44', '45\n' ]
```

Tutaj metodę łańcuchów `split` wykorzystaliśmy do pocięcia wiersza w miejscu wystąpienia przecinków. W rezultacie otrzymujemy listę podłańcuchów zawierających poszczególne liczby. Gdybyśmy chcieli na nich wykonać jakieś działania matematyczne, nadal musielibyśmy przekształcić je z łańcuchów znaków na liczby.

```
>>> int(parts[1])                                # Konwersja z łańcucha na liczbę  
44  
>>> numbers = [int(P) for P in parts]           # Konwersja wszystkich liczb  
naraz  
>>> numbers  
[43, 44, 45]
```

Jak już wiemy, metoda `int` przekształca łańcuch cyfr w obiekt liczby całkowitej, a wyrażenie listy składanej z rozdziału 4. można wykorzystać do wywołania wszystkich elementów listy naraz (więcej informacji na temat list składanych znajdziesz się w dalszej części książki). Warto zauważyc, że nie musielibyśmy wywoływać metody `rstrip` w celu usunięcia znaku `\n` z końca ostatniego elementu — `int` i inne konwertery ignorują białe znaki znajdujące się wokół cyfr.

Wreszcie, by przekształcić listę oraz słownik z trzeciego wiersza pliku, możemy wywołać wbudowaną funkcję `eval`, która traktuje łańcuch jako fragment kodu wykonywalnego (z technicznego punktu widzenia: łańcuch znaków zawierający wyrażenie Pythona).

```
>>> line = F.readline()  
>>> line  
"[1, 2, 3]${'a': 1, 'b': 2}\n"  
>>> parts = line.split('$')                      # Podział na znaku $  
>>> parts  
[['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]  
>>> eval(parts[0])                             # Konwersja na dowolny typ  
obiektu  
[1, 2, 3]  
>>> objects = [eval(P) for P in parts]          # To samo w jednej liście  
>>> objects  
[[1, 2, 3], {'a': 1, 'b': 2}]
```

Ponieważ w rezultacie analizy składniowej i konwersji otrzymujemy listę normalnych obiektów Pythona w miejsce łańcuchów znaków, teraz możemy na tych obiektach zastosować zwykłe operacje na listach i słownikach.

Przechowywanie natywnych obiektów Pythona — moduł pickle

Zademonstrowane w ostatnim kodzie wykorzystanie funkcji `eval` do konwersji łańcuchów znaków na obiekty ma duże możliwości. Tak naprawdę — czasem nawet *za duże*. Funkcja `eval` z radością wykona dowolne wyrażenie Pythona, nawet takie, które skasuje wszystkie pliki z naszego komputera — pod warunkiem że ma do tego odpowiednie uprawnienia. Jeżeli jednak

naprawdę chcemy przechowywać obiekty Pythona, a nie możemy ufać źródłu danych z pliku, idealnie przyda nam się moduł `pickle` z biblioteki standardowej.

Moduł `pickle` jest zaawansowanym narzędziem pozwalającym na przechowywanie prawie każdego obiektu Pythona bezpośrednio w pliku bez konieczności dokonywania konwersji na łańcuch znaków i z niego. Jest bardzo ogólnym narzędziem do formatowania i przetwarzania danych. By na przykład przechować w pliku słownik, można to zrobić w sposób bezpośredni za pomocą modułu `pickle`.

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)                                # Serializacja dowolnego obiektu
w pliku
>>> F.close()
```

By później ponownie otrzymać słownik, wystarczy odtworzyć go raz jeszcze za pomocą `pickle`.

```
>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F)                                # Załadowanie dowolnego obiektu
z pliku
>>> E
{'a': 1, 'b': 2}
```

W ten sposób z powrotem otrzymamy równoważny obiekt słownika, bez konieczności przeprowadzania żadnych ręcznych podziałów czy konwersji. Moduł `pickle` wykonuje operację znaną jako *serializacja obiektów* — konwersja obiektów na łańcuchy bajtów i z powrotem. Jego zaletą jest to, że wymaga niewiele pracy z naszej strony. Moduł ten wewnętrznie tłumaczy nasz słownik na postać łańcucha znaków, choć nie wygląda to zbyt ciekawie przy odczycie (i może się jeszcze różnić przy użyciu innych trybów protokołu danych).

```
>>> open('datafile.pkl', 'rb').read()
b'\x80\x03}q\x00(X\x01\x00\x00\x00bq\x01K\x02X\x01\x00\x00\x00aq\x02K\x01u.'
```

Ponieważ `pickle` potrafi zrekonstruować obiekt z tego formatu, nie musimy się tym zajmować samodzielnie. Więcej informacji na temat modułu `pickle` można znaleźć w dokumentacji biblioteki standardowej Pythona lub importując `pickle` i przekazując ten moduł do funkcji `help` w sesji interaktywnej. Przy okazji warto również zapoznać się z modułem `shelve`. Jest to narzędzie wykorzystujące moduł `pickle` do przechowywania obiektów Pythona w systemie plików dostępnych po kluczu, co znacznie wykracza poza omawiane tu zagadnienia. Przykład działania modułu `shelve` zobaczymy jednak w rozdziale 28., natomiast inne przykłady zastosowania modułu `pickle` — w rozdziałach 31. oraz 37.



Zwróc uwagę, że plik wykorzystywany do przechowywania zserializowanego obiektu otwieramy w *trybie binarnym*: tryb ten jest zawsze wymagany w Pythonie 3.x, ponieważ moduł `pickle` tworzy i wykorzystuje obiekt łańcucha znaków `bytes`. Obiekty tego typu wymuszają pliki w trybie binarnym (w wersji 3.0 pliki w trybie tekstowym wymuszają z kolei łańcuchy znaków `str`). We wcześniejszych wersjach Pythona użycie plików w trybie tekstowym dla protokołu 0 (domyślnego, tworzącego tekst ASCII) było w porządku, o ile tryb ten był wykorzystywany w sposób spójny. Wyższe protokoły wymagały plików w trybie binarnym. Domyślnym protokołem Pythona 3.x jest protokół 3 (binarny), jednak obiekt `bytes` wykorzystywany jest nawet dla protokołu 0.Więcej

informacji na ten temat można znaleźć w rozdziale 28., rozdziale 31. i rozdziale 37., dokumentacji biblioteki standardowej Pythona lub innych książkach.

Python 2.x zawiera również moduł cPickle, będący zoptymalizowaną wersją modułu pickle; można go importować w sposób bezpośredni w celu zwiększenia szybkości działania programu. W Pythonie 3.x nazwa tego modułu została zmieniona na _pickle; jest on wykorzystywany automatycznie przez pickle — skrypty po prostu importują moduł pickle i pozwalają Pythonowi na jego samodzielna optymalizację.

Przechowywanie obiektów Pythona w formacie JSON

Moduł pickle, przedstawiony w poprzedniej sekcji, tłumaczy niemal dowolne obiekty Pythona na zastrzeżony format opracowany specjalnie dla Pythona i doskonalony pod kątem wydajności przez wiele lat. JSON to nowszy i coraz bardziej popularny format wymiany danych, który jest niezależny od języka programowania i obsługiwany przez różne systemy. Na przykład *baza danych MongoDB* przechowuje dane w postaci dokumentów JSON (wykorzystując binarną wersję formatu JSON).

JSON nie obsługuje tak szerokiego zakresu typów obiektów Pythona jak pickle, ale jego przenośność jest zaletą w wielu zastosowaniach i stanowi inny sposób serializacji wybranych obiektów Pythona do przechowywania i przesyłania. Ponadto, ponieważ składnia formatu JSON jest mocno zbliżona do słowników i list Pythona, tłumaczenie do i z obiektów Pythona jest banalne i jest zautomatyzowane przez standardowy moduł biblioteki json.

Na przykład słownik Pythona ze strukturami zagnieźdzonymi jest bardzo podobny do danych JSON, chociaż zmienne i wyrażenia Pythona obsługują bogatsze opcje strukturyzacji (każde z poniższych poleceń może być dowolnym wyrażeniem w kodzie Pythona):

```
>>> name = dict(first='Adam', last='Kowalski')
>>> rec = dict(name=name, job=['dev', 'mgr'], age=40.5)
>>> rec
{'job': ['dev', 'mgr'], 'name': {'last': 'Kowalski', 'first': 'Adam'}, 'age': 40.5}
```

Ostateczny wygląd słownika pokazany w powyższym przykładzie jest prawidłowym literałem w kodzie Pythona i po wyświetleniu wygląda niemal jak JSON, ale użycie modułu json sprawia, że konwersja formatu staje się oficjalna — w przykładzie poniżej zamieniamy obiekty Pythona do i z serializowanej postaci JSON:

```
>>> import json
>>> json.dumps(rec)
'{"job": ["dev", "mgr"], "name": {"last": "Kowalski", "first": "Adam"}, "age": 40.5}'
>>> S = json.dumps(rec)
>>> S
'{"job": ["dev", "mgr"], "name": {"last": "Kowalski", "first": "Adam"}, "age": 40.5}'
>>> O = json.loads(S)
>>> O
```

```
{'job': ['dev', 'mgr'], 'name': {'last': 'Kowalski', 'first': 'Adam'}, 'age': 40.5}  
>>> 0 == rec  
True
```

Podobnie łatwo jest tłumaczyć obiekty Pythona na format JSON i na odwrót w plikach. Przed zapisaniem w pliku Twoje dane są po prostu obiektami Pythona; moduł JSON odtwarza je z reprezentacji tekstowej JSON po załadowaniu go z pliku:

```
>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)  
>>> print(open('testjson.txt').read())  
{  
    "job": [  
        "dev",  
        "mgr"  
    ],  
    "name": {  
        "last": "Kowalski",  
        "first": "Adam"  
    },  
    "age": 40.5  
}  
>>> P = json.load(open('testjson.txt'))  
>>> P  
{'job': ['dev', 'mgr'], 'name': {'last': 'Kowalski', 'first': 'Adam'}, 'age': 40.5}
```

Po przetłumaczeniu z postaci JSON dane są przetwarzane przy użyciu normalnych operacji obiektowych w skrypcie języka Python. Aby uzyskać więcej informacji na tematy związane z JSON, powinieneś zjrzeć do podręcznika biblioteki Pythona i przeszukać dostępne źródła w internecie.

Zauważ, że w plikach JSON wszystkie ciągi znaków są zapisane w *Unicode*, aby obsługiwać teksty zawierające znaki z zestawów międzynarodowych, więc po odczytaniu takiego tekstu w Pythonie 2.x (ale nie w 3.x) zobaczysz na początku wiodący znak u; wynika to wprost ze składni obiektów Unicode w wersji 2.x, o czym wspominaliśmy w rozdziałach 4. i 7., a omówimy w całości w rozdziale 37. Ponieważ ciągi tekstowe Unicode obsługują wszystkie standardowe operacje na łańcuchach, gdy tekst znajduje się w pamięci, różnica w kodzie jest nieznaczna i zaczyna mieć znaczenie dopiero przy przesyłaniu tekstu do i z plików oraz w przypadku tekstów innych niż ASCII, gdzie musimy brać pod uwagę sposób kodowania.



W świecie Pythona istnieje także wsparcie dla tłumaczenia obiektów na format XML i z formatu XML, czyli formatu tekowego, którego będziemy używać w rozdziale 37.; więcej szczegółowych informacji na temat tego formatu znajdziesz w internecie.

Aby dowiedzieć się czegoś więcej na temat innego, częściowo powiązanego narzędzia, które obsługuje sformatowane pliki danych, powinieneś zjrzeć do

dokumentacji modułu csv Pythona, która pozwala na tworzenie, analizowanie i przetwarzanie danych zapisanych plikach i ciągach znaków w formacie CSV (ang. *Comma-separated values*; wartości oddzielone przecinkami). Taki format nie pozwala wprawdzie na bezpośrednią serializację i deserializację obiektów Pythona, ale jest innym, bardzo rozpowszechnionym formatem wymiany danych:

```
>>> import csv  
>>> rdr = csv.reader(open('csvdata.txt'))  
>>> for row in rdr: print(row)  
...  
['a', 'bbb', 'cc', 'ddd']  
['11', '22', '33', '44']
```

Przechowywanie spakowanych danych binarnych – moduł struct

Jeszcze jedna uwaga związana z plikami, zanim przejdziemy dalej: niektóre zaawansowane aplikacje mają również do czynienia ze spakowanymi danymi binarnymi, utworzonymi na przykład przez program napisany w języku C czy połączenie sieciowe. Biblioteka standardowa Pythona zawiera narzędzie pomocne w takiej sytuacji — moduł struct potrafi zarówno tworzyć, jak i przetwarzać spakowane dane binarne. W pewnym sensie jest on kolejnym narzędziem do konwersji danych, które interpretuje łańcuchy znaków w plikach jako dane binarne.

O tym narzędziu wspominaliśmy już co prawda w rozdziale 4., ale wracamy do niego, aby spojrzeć na nie z nieco innej perspektywy. Na przykład, aby utworzyć plik ze spakowanymi danymi binarnymi, należy otworzyć go w trybie 'wb' (zapis binarny) i przekazać do struct formatujący łańcuch znaków wraz z obiektami Pythona. Zastosowany poniżej formatujący łańcuch znaków oznacza spakowanie danych w postaci czterobajtowej liczby całkowitej, czterożnakowego łańcucha (który począwszy od wersji 3.2 musi być typu bytes) i dwubajtowej liczby całkowitej, gdzie wszystkie elementy zapisane są z kolejnością bajtów big-endian [2] (inne kody formatów pozwalają na dopełnianie bajtami, obsługę liczb zmiennoprzecinkowych itp.).

```
>>> F = open('data.bin', 'wb') # Otwarcie pliku binarnego do zapisu  
>>> import struct  
>>> data = struct.pack('>i4sh', 7, b'jajo', 8) # Utworzenie spakowanych danych binarnych  
>>> data  
\x00\x00\x00\x07jajo\x00\x08  
>>> F.write(data) # Zapisanie łańcucha bajtowego  
>>> F.close()
```

Python tworzy binarny łańcuch danych bytes, który zapisujemy do pliku w normalny sposób (składa się on głównie ze znaków niedrukowanych zapisanych w szesnastkowych sekwencjach ucieczki). By przetworzyć te wartości na normalne obiekty Pythona, wystarczy z powrotem wczytać łańcuch i rozpakować go z użyciem tego samego formatującego łańcucha znaków.

Python dokonuje ekstrakcji wartości na normalne obiekty Pythona (liczby całkowite i łańcuch znaków).

```
>>> F = open('data.bin', 'rb')
>>> data = F.read()                                # Pobranie spakowanych danych
binarnych
>>> data
'\x00\x00\x00\x07jajo\x00\x08'
>>> values = struct.unpack('>i4sh', data)      # Konwersja na obiekty Pythona
>>> values
(7, b'jajo', 8)
```

Pliki z danymi binarnymi są zaawansowanymi narzędziami niskiego poziomu, których nie będziemy tutaj omawiali bardziej szczegółowo. Więcej informacji na ich temat można znaleźć w omówieniu modułu `struct` w rozdziale 37., a także w dokumentacji biblioteki standardowej Pythona lub po zimportowaniu modułu `struct` i interaktywnym przekazaniu go do funkcji `help`. Warto również zauważyć, że binarne tryby przetwarzania plików '`wb`' i '`rb`' można wykorzystać do przetwarzania prostych plików binarnych, takich jak obrazki czy pliki dźwiękowe, jako całości bez konieczności rozpakowywania ich zawartości.

Menedżery kontekstu plików

Warto również zwrócić uwagę na omówienie zagadnień związanych z *menedżerami kontekstu* plików, które znajdziesz w rozdziale 34.; jest to nowość, która została wprowadzona w Pythonie w wersjach 3.0 i 2.6. Choć jest to raczej opcja dziedziny przetwarzania wyjątków niż samych plików, umożliwia opakowanie kodu przetwarzającego pliki w warstwę logiki pozwalającą dopilnować, że plik po zakończeniu użycia będzie automatycznie zamknięty (a w razie potrzeby jego buforowana zawartość zostanie przeniesiona na dysk) — nie musimy więc polegać na mechanizmie automatycznego zamknięcia podczas czyszczenia pamięci.

```
with open(r'C:\code\data.txt') as myfile:    # Więcej informacji w rozdziale
34.

    for line in myfile:
        ...tutaj przetwarzamy wiersz...
```

Omówiona w rozdziale 34. instrukcja `try/finally` udostępnia podobną funkcjonalność, jednak kosztem dodatkowego kodu — a dokładniej mówiąc, trzech dodatkowych wierszy. Często możemy jednak uniknąć obu rozwiązań i pozwolić Pythonowi na automatyczne zamknięcie plików.

```
myfile = open(r'C:\code\data.txt')

try:
    for line in myfile:
        ...tutaj przetwarzamy wiersz...
finally:
    myfile.close()
```

Schemat kodu z menedżerem kontekstu `with` zapewnia uwolnienie zasobów systemowych we wszystkich wersjach Pythona i może być bardzo użytecznym rozwiązaniem dla plików

wyjściowych, gwarantującym odpowiednie opróżnianie bufora; w przeciwnieństwie do bardziej ogólnej konstrukcji z poleceniem `try` menedżer kontekstu jest jednak ograniczony do obiektów obsługujących jego protokół. Ponieważ oba powyższe rozwiązania wymagają nieco dokładniejszego przedstawienia, ich szczegółowe omówienie odłożymy do dalszej części książki.

Inne narzędzia powiązane z plikami

Istnieją również dodatkowe, bardziej wyspecjalizowane metody plików zaprezentowane w tabeli 9.2, a nawet takie, których w tej tabeli nie ma. Wspomniana wcześniej metoda `seek` ponownie ustawia aktualną pozycję w pliku (kolejna operacja odczytu lub zapisu zacznie się w tym miejscu), a metoda `flush` wymusza zapisanie zbuforowanych danych wyjściowych na dysku bez zamknięcia połączenia (domyślnie pliki zawsze są buforowane).

Kompletną listę metod plików można znaleźć w dokumentacji biblioteki standardowej Pythona oraz w książkach wymienionych w „Przedmowie”; aby szybko wyświetlić tekst pomocy, możesz również wywołać interaktywnie funkcje `dir` bądź `help`, przekazując im otwarty obiekt pliku (w Pythonie 2.x, ale już nie w 3.x, można także przekazać nazwę `file`). Więcej przykładów przetwarzania plików można znaleźć w ramce „Warto pamiętać — skanery plików” w rozdziale 13. Przedstawia ona często używane wzorce kodu pętli skanujących pliki, zawierające instrukcje, których jeszcze nie omawialiśmy.

Warto również zauważyć, że choć funkcja `open` oraz zwracane przez nią obiekty plików są naszym głównym interfejsem do plików zewnętrznych ze skryptów Pythona, w zestawie narzędzi Pythona istnieją również inne sposoby. Dostępne są na przykład:

Strumienie standardowe

Wstępnie otwarte obiekty plików w module `sys`, takie jak `sys.stdout` (więcej informacji na ten temat znajdziesz w podrozdziale „Operacje drukowania” w rozdziale 11.).

Pliki deskryptorów z modułu os

Uchwyty plików, które pozwalają na wykorzystywanie narzędzi niższego poziomu, takich jak blokowanie plików (zobacz także tryb „x” funkcji `open` w Pythonie 3.3, pozwalający na tzw. wyłączne tworzenie plików).

Gniazda, potoki, kolejki FIFO

Obiekty podobne do plików, wykorzystywane do synchronizacji procesów lub komunikacji za pośrednictwem sieci.

Pliki z dostępem według kluczy z modułu shelve

Wykorzystywane do bezpośredniego przechowywania niezmodyfikowanych, serializowanych obiektów Pythona według kluczy (używamy ich w rozdziale 28.).

Strumienie poleceń powłoki

Narzędzia takie jak `os.popen` czy `subprocess.Popen`, obsługujące wywoływanie poleceń powłoki oraz wczytywanie ich strumieni standardowych oraz zapisywanie do nich (przykłady takich rozwiązań znajdziesz w rozdziałach 13. i 21.).

W domenie `open source` można znaleźć jeszcze więcej narzędzi do pracy z plikami, w tym narzędzia obsługujące komunikację z portami szeregowymi za pomocą rozszerzenia `PySerial` czy programy interaktywne, takie jak system `pexpect`. Więcej szczegółowych informacji na ten temat znajdziesz w internecie.

Uwaga na temat wersji: W Pythonie 2.x wbudowana funkcja `open` jest właściwie synonimem funkcji `file`, stąd technicznie rzecz biorąc, pliki można otwierać za pomocą wywołania albo `open`, albo `file` (choć w przypadku otwierania



preferowanym rozwiązaniem jest użycie funkcji `open`). W Pythonie 3.x funkcja `file` nie jest już dostępna z powodu pokrywania się jej funkcjonalności z funkcją `open`.

Użytkownicy Pythona 2.x mogą także używać nazwy `file` jako typu obiektu pliku w celu dostosowania plików do własnych potrzeb za pomocą programowania zorientowanego obiektywo (opisanego w dalszej części książki). W Pythonie 3.x pliki radykalnie się zmieniły. Klasy wykorzystywane do implementacji obiektów plików znajdują się w module biblioteki standardowej `io`. Więcej szczegółowych informacji na temat plików znajdziesz w dokumentacji tego modułu lub kodzie klas udostępnianych przez niego na potrzeby dostosowania do własnych potrzeb, a także po wywołaniu polecenia `type(F)` dla otwartego pliku `F`.

Przegląd i podsumowanie podstawowych typów obiektów

Po zapoznaniu się ze wszystkimi podstawowymi typami wbudowanymi Pythona warto zakończyć ich omawianie podsumowaniem ich niektórych wspólnych właściwości. W tabeli 9.3 sklasyfikowano wszystkie najważniejsze omówione typy obiektów zgodnie z wprowadzonymi wcześniej kategoriami typów. Oto kilka uwag ogólnych, o których warto pamiętać:

Tabela 9.3. Klasyfikacja typów obiektów

Typ obiektu	Kategoria typu	Zmienny?
Liczby (wszystkie)	Liczbowy	Nie
Łańcuchy znaków (wszystkie)	Sekwencja	Nie
Listy	Sekwencja	Tak
Słowniki	Odwzorowanie	Tak
Krotki	Sekwencja	Nie
Pliki	Rozszerzenie	Nie dotyczy
Zbiory	Zbiór	Tak
Frozenset	Zbiór	Nie
bytearray	Sekwencja	Tak

- Obiekty współdzielą operacje zgodnie z kategoriami. Na przykład obiekty sekwencyjne, takie jak ciągi znaków, listy i krotki, współdzielą operacje na sekwencjach, takie jak konkatenacja, obliczanie długości czy indeksowanie.
- Modyfikowane w miejscu mogą być jedynie obiekty mutowalne (listy, słowniki oraz zbiory); nie można tego robić w przypadku liczb, łańcuchów znaków czy krotek.
- Pliki jedynie eksportują metody, dlatego kwestia mutowalności ich nie dotyczy. Co prawda ich stan może się zmieniać podczas przetwarzania, ale nie ma to zbyt wiele wspólnego z ograniczeniami związanymi z mutowalnością bądź niemutowalnością poszczególnych typów obiektów Pythona.

- „Liczby” z tabeli 9.3 obejmują wszystkie typy liczb: całkowite (oraz odrębny typ długiej liczby całkowitej z Pythona 2.x), zmiennoprzecinkowe, zespolone, dziesiętne oraz ułamkowe.
- „Łańcuchy znaków” z tabeli 9.3 obejmują typ `str`, a także `bytes` z Pythona 3.x oraz `unicode` z wersji 2.x. Typ `bytearray` dostępny w wersjach 3.x, 2.6 i 2.7 jest mutowalny.
- Zbiory przypominają nieco klucze słownika bez wartości, jednak nie odwzorowują wartości i nie są uporządkowane, dlatego zbiory nie są ani odwzorowaniem, ani sekwencją. Typ `frozenset` to niemutowalny wariant typu `set`.
- Poza operacjami kategorii typu w Pythonie 2.6 oraz 3.0, wszystkie typy z tabeli 9.3 posiadają wywoływalne metody, które są zazwyczaj specyficzne dla ich typu. Elastyczność obiektów

Warto pamiętać — przeciążanie operatorów

W szóstej części książki zobaczymy, że obiekty implementowane za pomocą klas mogą w dowolny sposób korzystać z tych kategorii. Jeżeli na przykład chcemy udostępnić nowy rodzaj wyspecjalizowanego obiektu sekwencji, spójnego z wbudowanymi sekwencjami, wystarczy zakodować klasę przeciążającą na przykład indeksowanie i konkatenację.

`class MySequence:`

```
def __getitem__(self, index):
    # Wywoływany na self[index], others
def __add__(self, other):
    # Wywoływany na self + other
def __iter__(self):
    # Preferowany w iteracjach
```

Nowy obiekt może również być mutowalny lub niemutowalny, w zależności od wyboru metod wywoływanych dla operacji zmieniających obiekt w miejscu (na przykład `__setitem__` jest wywoływane na przypisaniach `self[index] = value`). Choć wykracza to poza tematykę niniejszej książki, można również implementować nowe obiekty w językach zewnętrznych, jak C — jako rozszerzenia języka C. W tym przypadku wypełnia się wskaźniki do funkcji C w celu wyboru spośród zbioru operacji na liczbach, sekwencjach i odwzorowaniach.

Elastyczność obiektów

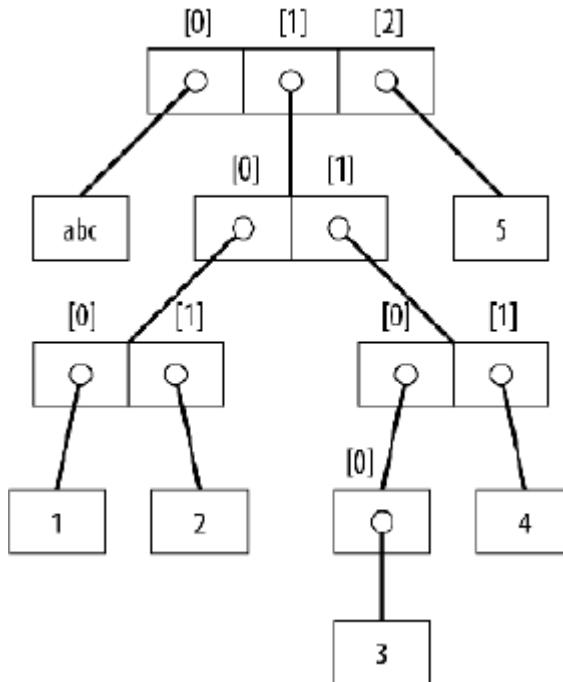
W tej części książki wprowadziliśmy kilka złożonych typów obiektów — kolekcje zawierające komponenty. Ogólnie rzecz biorąc:

- listy, słowniki oraz krotki mogą przechowywać dowolne obiekty,
- zbiory mogą zawierać dowolne rodzaje obiektów niemutowalnych,
- listy, słowniki i krotki można dowolnie zagnieżdżać,
- listy, słowniki oraz zbiory mogą dynamicznie rosnąć lub się kurczyć.

Ponieważ złożone typy obiektów Pythona obsługują dowolne struktury, świetnie nadają się do reprezentowania skomplikowanych informacji w programach. Na przykład wartości słownika mogą być listami zawierającymi krotki, które z kolei zawierają słowniki — i tak dalej. Zagnieżdżanie może mieć dowolną głębokość niezbędną do przedstawienia danych, które mają być przetworzone.

Przyjrzyjmy się przykładowi zagnieżdżania. Poniższy listing definiuje drzewo zagnieżdżonych złożonych obiektów sekwencji, zaprezentowanych również na rysunku 9.1. Aby uzyskać dostęp

do poszczególnych komponentów, można skorzystać z dowolnej liczby operacji indeksowania. Python oblicza indeksy od lewej do prawej strony i z każdym krokiem pobiera referencje do głębszej zagnieżdżonych obiektów. Rysunek 9.1 może przedstawać patologicznie złożoną strukturę danych, jednak dobrze ilustruje składnię wykorzystywaną w celu uzyskania dostępu do obiektów zagnieżdżonych.



Rysunek 9.1. Drzewo zagnieżdżonych obiektów wraz z wartościami przesunięcia komponentów, utworzone dzięki wykonaniu literała `['abc', [(1, 2), ([3], 4)], 5]`. Obiekty zagnieżdżone z punktu widzenia składni są wewnętrznie reprezentowane przez referencje (wskaźniki) do osobnych fragmentów pamięci

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

Referencje a kopie

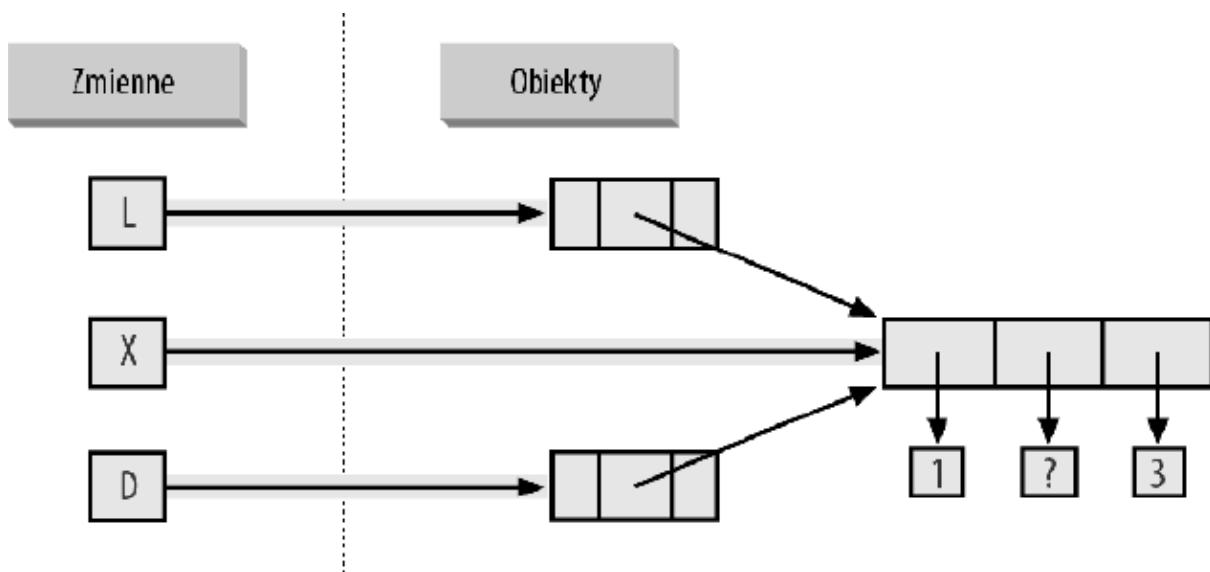
W rozdziale 6. wspominaliśmy, że przypisanie zawsze przechowuje referencje do obiektów, a nie ich kopie. W praktyce najczęściej o to nam chodzi. Ponieważ jednak przypisania mogą

wygenerować wiele referencji do tego samego obiektu, należy być świadomym, że modyfikacja zmiennego obiektu w miejscu może wpływać na inne referencje do tego samego obiektu znajdujące się w programie. Jeżeli nie życzymy sobie takiego zachowania, będziemy musieli nakazać Pythonowi utworzyć kopię obiektu.

Z tym zjawiskiem zapoznaliśmy się już w rozdziale 6., jednak staje się ono bardziej subtelne, kiedy w grę wchodzą większe obiekty, z którymi już do tej pory mieliśmy do czynienia. Poniższy przykład tworzy listę przypisaną do zmiennej X i kolejną listę przypisaną do zmiennej L z osadzoną referencją do listy X. Tworzy również słownik D zawierający kolejną referencję do listy X.

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']                                     # Osadzone referencje do
   zmiennej X
>>> D = {'x':X, 'y':2}
```

W tym momencie istnieją trzy referencje do pierwszej utworzonej listy — ze zmiennej X, ze środka listy przypisanej do zmiennej L oraz ze środka słownika przypisanego do zmiennej D. Sytuacja ta została przedstawiona na rysunku 9.2.



Rysunek 9.2. Współdzielone referencje do obiektu. Ponieważ referencje do listy, do której odnosi się zmenna X, znajdują się również wewnątrz obiektów, do których odnoszą się zmienne L i D, zmiana współdzielonej listy z X sprawia, że lista ta zmieni się również w L i D

Ponieważ listy są mutowalne, modyfikacja współdzielonego obiektu listy w dowolnej z trzech referencji zmieni ją również dla dwóch pozostałych referencji.

```
>>> X[1] = 'niespodzianka'                                # Zmienia wszystkie trzy
   referencje!
>>> L
['a', [1, 'niespodzianka', 3], 'b']
>>> D
{'x': [1, 'niespodzianka', 3], 'y': 2}
```

Referencje są wysokopoziomową analogią do wskaźników z innych języków programowania. Choć nie możemy uzyskać dostępu do samej referencji, możliwe jest przechowanie tej samej referencji w większej liczbie miejsc (zmiennych czy list). Umożliwia to na przykład przekazywanie większego obiektu w programie bez konieczności kosztownego generowania przy tym kopii. Jeżeli jednak naprawdę zależy nam na kopii, możemy ich zażądać.

- Wyrażenia wycinków z pustymi granicami (`L[:]`) kopią sekwencje.
- Metoda `copy(D.copy())` słowników, zbiorów oraz list kopiuje słownik, zbiór lub listę (metod `copy` listy jest dostępna od wersji 3.3).
- Niektóre wbudowane funkcje, takie jak `list` czy `dict` również tworzą kopie (`list(L)`, `dict(D)`, `set(S)`).
- Moduł `copy` biblioteki standardowej w razie potrzeby może tworzyć pełne kopie.

Na przykład założmy, że mamy listę oraz słownik i nie chcemy, aby ich wartości zmieniały się za pośrednictwem innych zmiennych.

```
>>> L = [1, 2, 3]
>>> D = {'a':1, 'b':2}
```

Aby temu zapobiec, wystarczy przypisać do innych zmiennych kopie, a nie referencje do tych samych obiektów.

```
>>> A = L[:]                                # Zamiast A = L (lub list(L))
>>> B = D.copy()                            # Zamiast B = D (tak samo dla
                                             zbiorów)
```

W ten sposób zmiany wprowadzone w innych zmiennych będą modyfikowały kopie, a nie oryginalne obiekty.

```
>>> A[1] = 'Ni'
>>> B['c'] = 'mielonka'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'mielonka', 'b': 2})
```

W przypadku naszego początkowego przykładu możemy uniknąć efektów ubocznych referencji, sporządzając wycinek z oryginalnej listy zamiast podawania nazwy zmiennej.

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']                      # Osadzenie kopii obiektu
                                             zmiennej X
>>> D = {'x':X[:], 'y':2}
```

To zmienia obraz sytuacji z rysunku 9.2. Zmienne `L` i `D` będą teraz wskazywały na inne listy niż `X`. Rezultat będzie taki, że zmiany wprowadzone za pośrednictwem zmiennej `X` będą miały wpływ wyłącznie na `X`, a nie na `L` i `D`. W podobny sposób modyfikacje `L` i `D` nie wpłyną na `X`.

Ostatnia uwaga dotycząca kopii: wycinki z pustymi granicami i metoda `copy` słownika wykonują jedynie kopie *najwyższego poziomu*. Oznacza to, że nie kopią one zagnieżdżonych struktur danych, jeżeli takie są obecne. Jeżeli potrzebujesz kompletnej, w pełni niezależnej kopii głęboko zagnieżdzonej struktury danych (takiej jak różne struktury rekordów, które kodowaliśmy w

ostatnich rozdziałach), powinieneś skorzystać ze standardowego modułu `copy`, przedstawionego w rozdziale 6.:

```
import copy  
X = copy.deepcopy(Y)                      # Pełna kopia dowolnie zagnieżdżonego  
obiektu Y
```

Wywołanie to przechodzi przez obiekt w sposób rekurencyjny w celu skopiowania wszystkich jego części. Jest to jednak o wiele rzadszy przypadek (i dlatego do zaimplementowania go potrzeba nieco więcej kodu). To referencje są zwykle tym, czego potrzebujesz; gdy tak nie jest, metody wycinania i kopowania zazwyczaj kopią dokładnie tyle, ile jest potrzebne.

Porównania, testy równości i prawda

Wszystkie obiekty Pythona poddają się również porównaniom — testom równości, względnej wielkości itd. Porównania w Pythonie zawsze sprawdzają wszystkie części obiektów złożonych, dopóki nie zostanie ustalony wynik. Tak naprawdę, kiedy obecne są obiekty zagnieżdżone, Python automatycznie przechodzi struktury danych w celu zastosowania porównań w sposób rekurencyjny od lewej do prawej strony i tak głęboko, jak jest to konieczne. Pierwsza napotkana różnica przesąduje o wyniku porównania.

Jest to czasami nazywane *porównaniem rekurencyjnym* — to samo porównanie wymagane na obiektach najwyższego poziomu jest stosowane do każdego z zagnieżdżonych obiektów, następnie dla każdego zagnieżdżonego w nich obiektu, i tak dalej, aż do znalezienia wyniku. W dalszej części tej książki, w rozdziale 19., pokażemy, jak pisać własne funkcje rekurencyjne, które działają podobnie na strukturach zagnieżdżonych. Na razie, myśląc o porównywaniu takich struktur, możesz sobie to wyobrażać jako operację podobną do porównywania wszystkich połączonych ze sobą stron w dwóch witrynach internetowych, gdzie możemy to zrobić za pomocą odpowiednio przygotowanej funkcji rekurencyjnej.

W przypadku podstawowych typów obiektów Pythona taka rekurencja jest implementowana automatycznie. Na przykład porównywanie obiektów listy automatycznie zestawia wszystkie ich komponenty do momentu znalezienia pierwszej niezgodności lub osiągnięcia końca listy:

```
>>> L1 = [1, ('a', 3)]                                # Ta sama wartość, unikalne  
obiekty  
>>> L2 = [1, ('a', 3)]  
>>> L1 == L2, L1 is L2                               # Odpowiednik? Ten sam obiekt?  
(True, False)
```

W powyższym kodzie do zmiennych `L1` i `L2` przypisano listy będące odpowiednikami, jednak różnymi obiektami. Ze względu na naturę referencji Pythona (przedstawioną w rozdziale 6.) równość można sprawdzać na dwa sposoby.

- **Operator `==` sprawdza równość wartości.** Python wykonuje test równości, porównując rekurencyjnie wszystkie zagnieżdżone obiekty.
- **Operator `is` sprawdza identyczność obiektów.** Python sprawdza, czy dwa podane obiekty są tak naprawdę jednym i tym samym obiektem (to znaczy znajdują się pod jednym adresem w pamięci).

W poprzednim przykładzie listy `L1` i `L2` zdają test `==` (mają takie same wartości, ponieważ wszystkie ich komponenty są takie same), natomiast nie zdają testu `is` (są referencjami do dwóch różnych obiektów i tym samym dwóch różnych fragmentów pamięci). Warto jednak zwrócić uwagę, co dzieje się w przypadku krótkich łańcuchów znaków.

```
>>> S1 = 'mielonka'
```

```
>>> S2 = 'mielonka'  
>>> S1 == S2, S1 is S2  
(True, True)
```

Tutaj powinniśmy znowu mieć do czynienia z sytuacją, w której dwa odrębne obiekty mają tę samą wartość. Wynikiem testu `==` powinna być prawda, a testu `is` fałsz. Jednak ponieważ Python wewnętrznie umieszcza niektóre łańcuchy znaków w pamięci podręcznej i ponownie ich używa ze względu na optymalizację działania, tak naprawdę w pamięci istnieje tylko jeden łańcuch znaków: `'mielonka'`, wspólnie dzielony przez `S1` i `S2`. Z tego powodu test identyczności `is` zwraca `True`. By wywołać normalne zachowanie, musimy wykorzystać dłuższy łańcuch znaków.

```
>>> S1 = 'dłuższy łańcuch znaków'  
>>> S2 = 'dłuższy łańcuch znaków'  
>>> S1 == S2, S1 is S2  
(True, False)
```

Oczywiście ponieważ łańcuchy znaków są *niemutowalne*, mechanizm umieszczania obiektów w pamięci podręcznej nie ma wpływu na nasz kod. łańcuchów znaków nie można modyfikować w miejscu bez względu na to, ile zmiennych się do nich odnosi. Jeżeli testy identyczności wydają Ci się mylące, powinieneś wrócić do rozdziału 6. i przypomnieć sobie wiadomości dotyczące referencji do obiektów.

Generalnie w prawie wszystkich testach równości wykorzystuje się operator `==`. Operator `is` zarezerwowany jest dla wysoce wyspecjalizowanych zadań. W dalszej części książki zobaczymy przykłady zastosowania tych operatorów.

Względne porównania wielkości są również stosowane rekurencyjnie do zagnieżdżonych struktur danych:

```
>>> L1 = [1, ('a', 3)]  
>>> L2 = [1, ('a', 2)]  
>>> L1 < L2, L1 == L2, L1 > L2          # Mniejszy, równy, większy –  
krotka wyników  
(False, False, True)
```

W powyższym kodzie lista `L1` jest większa od `L2`, ponieważ zagnieżdżone 3 jest większe od 2. Teraz powinieneś już wiedzieć, że wynik ostatniego wiersza kodu jest tak naprawdę krotką trzech obiektów — wyników trzech wpisanych wyrażeń (jest to przykład krotki bez nawiasów).

Python porównuje typy w następujący sposób:

- *Liczby* porównywane są zgodnie z ich względną wielkością po dokonaniu konwersji do najwyższego typu wspólnego (jeżeli to konieczne).
- *Łańcuchy znaków* porównywane są leksykograficznie (według kodów znaków zwracanych przez funkcję `ord`) oraz znak po znaku aż do znalezienia pierwszego niedopasowania różnicy bądź dotarcia do końca ciągu (`"abc" < "ac"`).
- *Listy i krotki* porównywane są poprzez zestawienie każdego komponentu od lewej do prawej strony oraz rekurencyjnie dla struktur zagnieżdżonych, aż do końca lub pierwszego niedopasowania (`[2] > [1, 2]`).
- *Zbiory* są równe, jeżeli oba zawierają te same elementy (formalnie, jeżeli każdy ze zbiorów jest podzbiorem drugiego), a porównania wielkości względnej zbiorów stosują testy podzbioru i nadzbioru.
- *Słowniki* uznawane są za równe, jeżeli ich posortowane listy (*klucz, wartość*) są równe. Porównania względnego rozmiaru nie są obsługiwane dla słowników w Pythonie 3.x, ale

- działają w wersji 2.x tak, jakby porównywały posortowane listy (*klucz, wartość*).
- Porównania mieszanych typów nieliczbowych (na przykład `1 < 'mielonka'`) są w Pythonie 3.x niedozwolone i powodują zgłoszenie błędu. Takie porównania są jednak dozwolone w Pythonie 2.x, jednak wykorzystują z góry określone reguły uporządkowania bazujące na nazwach. W sposób pośredni odnosi się to również do sortowania, które wewnętrznie wykorzystuje porównania. W Pythonie 3.x kolekcje nienumerycznych typów mieszanych nie mogą być sortowane.

Porównania obiektów ustrukturyzowanych wyglądają tak, jakbyśmy obiekty te zapisali jako literały i porównywali wszystkie ich elementy po jednym na raz od lewej do prawej strony. W kolejnych rozdziałach zobaczymy inne typy danych, które mogą zmienić sposób porównywania.

Porównywania i sortowania typów mieszanych w Pythonie 2.x i 3.x

Według ostatniego punktu na liście w poprzedniej sekcji zmiana w Pythonie 3.x dla porównań nienumerycznych typów mieszanych ma zastosowanie do testów *wielkości*, a nie równości, ale dotyczy również sortowania, które testy wielkości przeprowadzają wewnętrznie. W Pythonie 2.x działają wszystkie rodzaje porównań, chociaż typy mieszane porównywane są według dosyć arbitralnie ustalonego porządku:

```
c:\code> c:\python27\python
>>> 11 == '11'                                # Testy równości nie konwertują
wartości nienumerycznych
False
>>> 11 >= '11'                               # wersja 2.x porównuje typy mieszane
według nazw typów: int, str
False
>>> ['11', '22'].sort()                      # Podobnie jest z sortowaniem
>>> [11, '11'].sort()
```

Ale Python 3.x nie zezwala na testowanie rozmiarów typów mieszanych, za wyjątkiem typów numerycznych i typów ręcznie konwertowanych:

```
c:\code> c:\python33\python
>>> 11 == '11'                                # wersja 3.x: testy równości działają,
ale testy wielkości już nie
False
>>> 11 >= '11'
TypeError: unorderable types: int() > str()
>>> ['11', '22'].sort()                      # Podobnie jest z sortowaniem
>>> [11, '11'].sort()
TypeError: unorderable types: str() < int()
>>> 11 > 9.123                                # Mieszane typy numeryczne są
konwertowane do najwyższego typu
True
>>> str(11) >= '11', 11 >= int('11')      # Ręczna konwersja rozwiązuje problem
```

(True, True)

Porównywanie słowników w Pythonie 2.x i 3.x

Przedostatni przykład z poprzedniego podrozdziału również zasługuje na rozwinięcie. W Pythonie 2.x oraz wersjach wcześniejszych słowniki obsługują porównania rozmiaru, tak jakbyśmy porównywali posortowane listy par klucz/wartość:

```
C:\code> c:\python27\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2                                # Test równości słowników: wersje 2.x
+ 3.x
False
>>> D1 < D2                                # Test wielkości słowników: tylko
wersja 2.x
True
```

Jak wspominaliśmy krótko w rozdziale 8., w Pythonie 3.x porównania rozmiaru słowników zostały usunięte, ponieważ skutkują one często zbyt dużym narzutem na wydajność takiej operacji (testy równości w wersji 3.x wykorzystują zoptymalizowane rozwiązanie, które nie porównuje w sposób dosłowny posortowanych list par klucz/wartość):

```
C:\code> c:\python33\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
TypeError: unorderable types: dict() < dict()
```

Alternatywnym rozwiązaniem takich porównań w tej wersji Pythona jest napisanie pętli porównujących wartości według kluczy albo ręczne porównywanie posortowanych list par klucz/wartość — do tego celu wystarczy metoda `items` słownika oraz wbudowana funkcja `sorted`.

```
>>> list(D1.items())
[('b', 2), ('a', 1)]
>>> sorted(D1.items())
[('a', 1), ('b', 2)]
>>>
>>> sorted(D1.items()) < sorted(D2.items())          # Test wielkości w
wersji 3.x
True
>>> sorted(D1.items()) > sorted(D2.items())
False
```

Jak widać, takie rozwiązanie wymaga użycia nieco większej ilości kodu, ale w praktyce w większości programów korzystających z tego typu operacji implementowane są znacznie bardziej wydajne sposoby porównywania słowników zarówno w stosunku do rozwiązania pokazanego powyżej, jak i do oryginalnego mechanizmu działania porównań z Pythona 2.x.

Prawda czy fałsz, czyli znaczenie True i False w Pythonie

Warto zwrócić uwagę na to, że wyniki testów zwarcane w dwóch ostatnich przykładach reprezentują wartości prawdy i fałszu. Wyświetlane są one jako słowa `True` i `False`, jednak skoro już na poważnie używamy testów logicznych, warto wyjaśnić, co tak naprawdę znaczą te nazwy z formalnego punktu widzenia.

W Pythonie, jak w większości języków programowania, wartość `False` (fałsz) jest reprezentowana przez liczbę całkowitą 0, natomiast wartości `True` (prawda) odpowiadają liczba całkowita 1. Dodatkowo jednak Python rozpoznaje dowolną pustą strukturę danych jako fałsz, a dowolną niepustą strukturę danych jako prawdę. Mówiąc bardziej ogólnie, koncepcje prawdy i fałszu są wrodzonymi właściwościami każdego obiektu w Pythonie. Każdy obiekt jest albo prawdą, albo fałszem, zgodnie z poniższymi regułami:

- liczby są prawdą, jeżeli nie są zerem,
- pozostałe obiekty są prawdą, jeżeli są puste.

W tabeli 9.4 zaprezentowano przykłady prawdy i fałszu na bazie obiektów Pythona.

Tabela 9.4. Przykłady prawdy i fałszu w Pythonie

Obiekt	Wartość
"mielonka"	True
""	False
[1, 2]	True
[]	False
{'a':1}	True
{}	False
1	True
0.0	False
None	False

W jednym z zastosowań, ponieważ same obiekty mogą być albo prawdą, albo fałszem, programiści Pythona zwykle tworzą testy takie jak `if X:`, które zakładając, że `X` jest łańcuchem znaków, odpowiadają kodowi `if X != ''`. Innymi słowy, możesz przetestować sam obiekt, aby sprawdzić, czy ma jakąś zawartość, zamiast porównywać go z pustym, a zatem fałszywym obiektem tego samego typu (więcej szczegółowych informacji na temat instrukcji `if` znajdziesz w następnym rozdziale).

Obiekt `None`

Jak widać w ostatnim wierszu tabeli 9.4, Python udostępnia również specjalny obiekt `None`, który zawsze uznawany jest za fałsz. Obiekt `None` został krótko przedstawiony w rozdziale 4. Jest jedyną wartością specjalnego typu danych w Pythonie i zazwyczaj służy jako pusty obiekt zastępczy (podobnie jak wskaźnik `NULL` w języku C).

Na przykład spróbuj sobie przypomnieć, że w przypadku list nie możemy przypisać elementu do wartości przesunięcia, jeżeli taki offset jeszcze nie istnieje (lista nie może automagicznie rosnąć, jeżeli wykonasz przypisanie do elementu poza jej granicami). Aby wstępnie alokować listę składającą się ze stu elementów tak, aby można przypisywać wartości do dowolnego elementu listy, możesz ją wypełnić obiektami `None`.

```
>>> L = [None] * 100  
>>>  
>>> L  
[None, None, None, None, None, None, None, ..., ]
```

Powyższy kod nie ogranicza rozmiaru listy (nadal może ona później rosnąć lub się kurczyć), a po prostu ustawia jej początkową wielkość, tak by pozwolić na przyszłe operacje przypisania do indeksów. W ten sam sposób możemy oczywiście także zainicjalizować listę z serią zer, jednak praktyka wskazuje na użycie `None`, jeżeli zawartość listy nie jest jeszcze znana.

Pamiętaj jednak, że `None` nie oznacza „niezdefiniowany”. Inaczej mówiąc, `None` jest czymś, a nie niczym (mimo że nazwa tego obiektu wskazywałaby na coś innego!). `None` jest prawdziwym obiektem, zajmującym miejsce w pamięci i posiadającym wbudowaną nazwę otrzymaną od Pythona.

Zwróć uwagę na inne zastosowania tego specjalnego obiektu, o których będziemy mówić w dalszej części książki; jak się również przekonasz w części IV, jest to również domyślna wartość zwracana przez funkcje, które nie kończą pracy wykonaniem instrukcji `return` z wynikiem działania funkcji.

Typ `bool`

Warto również pamiętać, że typ Boolean Pythona (`bool`), wprowadzony w rozdziale 5., po prostu rozszerza koncepcje prawdy i fałszu w tym języku. Jak wiemy z rozdziału 5., wbudowane słowa kluczowe `True` oraz `False` są po prostu inaczej zapisanymi wersjami liczb całkowitych `1` i `0` — wygląda to trochę tak, jakby w całym Pythonie te dwa słowa zostały z góry przypisane do liczb `1` i `0`. Ze względu na sposób implementacji tego nowego typu jest to po prostu niewielkie rozszerzenie opisanych już pojęć prawdy i fałszu, wprowadzone w celu nadania tym wartościom lepiej rozpoznawalnego wyglądu.

- Kiedy w kodzie testów prawdy wykorzystamy te słowa w sposób jawnym, `True` i `False` są odpowiednikami liczb całkowitych `1` i `0`, które jednak w jaśniejszy sposób przedstawiają intencje programisty.
- Wyniki testów Boolean wykonywanych interaktywnie również wyświetlane są jako słowa `True` i `False`, a nie liczby całkowite `1` i `0`, by całość była bardziej zrozumiała.

W instrukcjach logicznych, takich jak `if`, nie musimy korzystać jedynie z typów Boolean. Wszystkie obiekty mają z natury przypisaną wartość prawdy lub fałszu i wszystkie koncepcje z wartościami Boolean opisane w niniejszym rozdziale będą działały, nawet jeżeli użyjemy innych typów. Python udostępnia funkcję wbudowaną `bool`, którą można wykorzystać do sprawdzenia wartości logicznej obiektu (to znaczy, czy ma on wartość `True` — a zatem jest niezerowy lub niepusty).

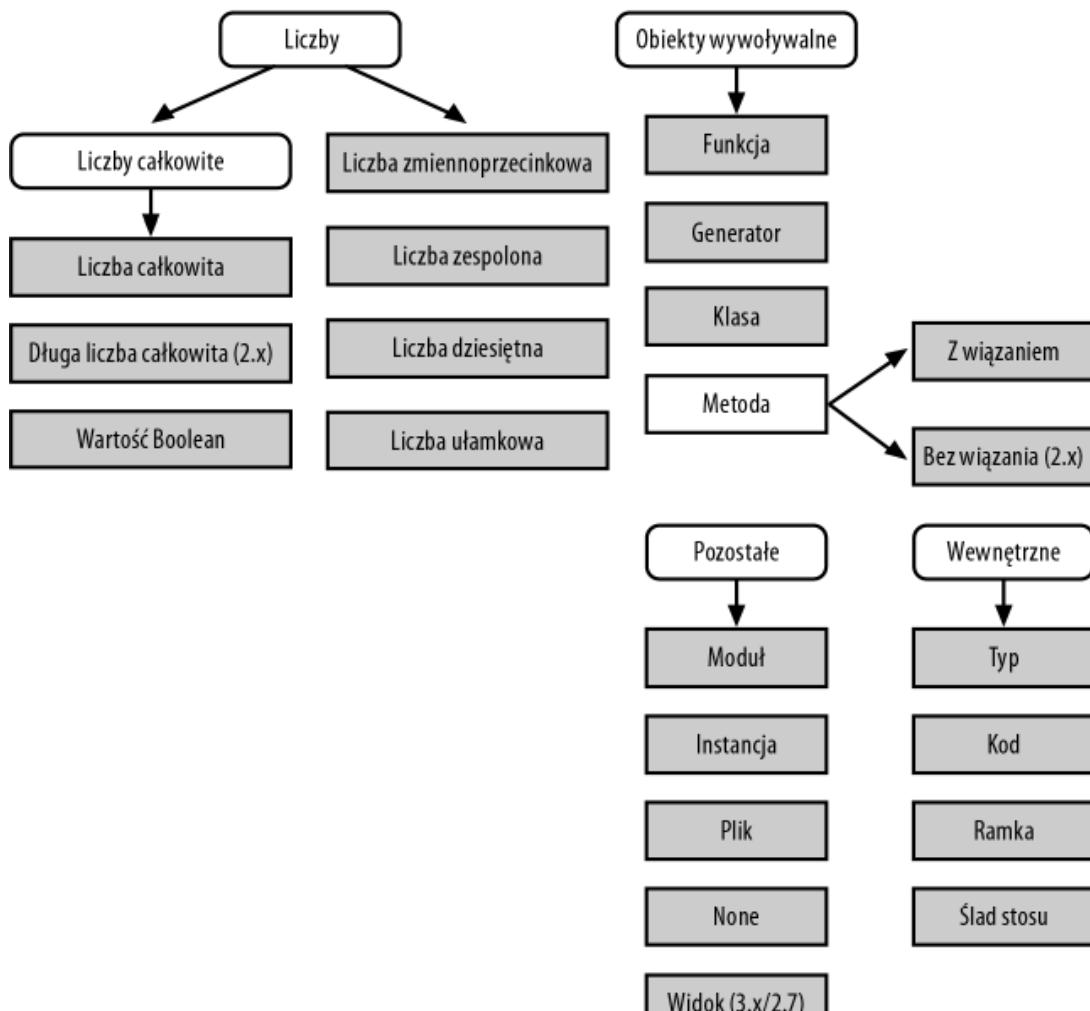
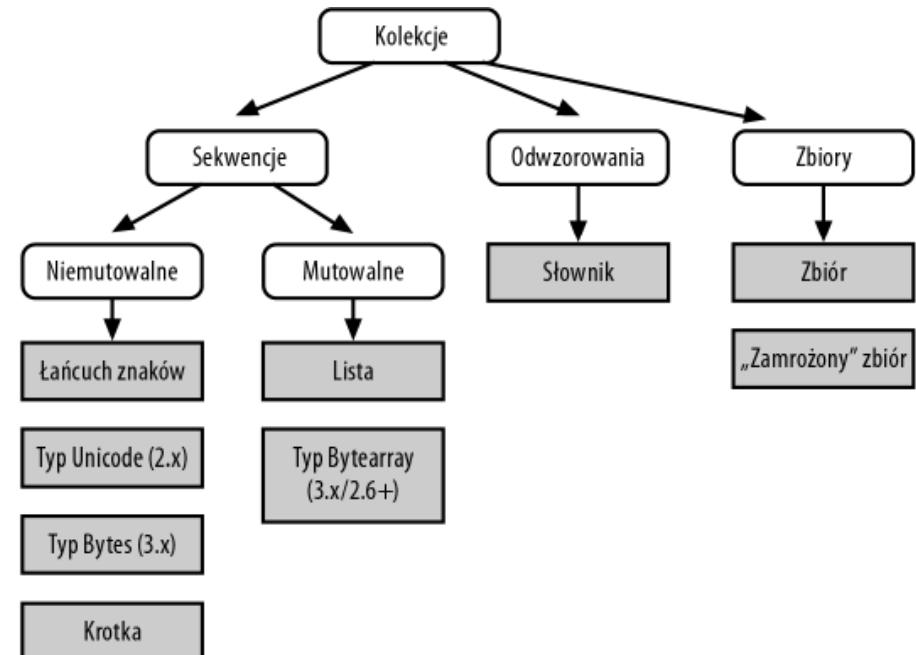
```
>>> bool(1)  
True
```

```
>>> bool('mielonka')
True
>>> bool({})
False
```

W praktyce rzadko zobaczymy typ Boolean zwracany przez testy logiki, ponieważ są one automatycznie wykorzystywane przez instrukcje `if` oraz inne narzędzia wyboru. Wartości typu Boolean będziemy jeszcze omawiać przy okazji rozważań nad instrukcjami logicznymi w rozdziale 12.

Hierarchie typów Pythona

Na rysunku 9.3 podsumowano wszystkie wbudowane typy obiektów dostępne w Pythonie wraz z ich wzajemnymi relacjami. Przyjrzelismy się najważniejszym z nich; większość innych typów obiektów z rysunku 9.3 odpowiada poszczególnym komponentom programu (jak funkcje i moduły) lub udostępnionym komponentom wewnętrznym interpretera (jak ramki stosu czy skompilowany kod).



Rysunek 9.3. Najważniejsze wbudowane typy obiektów Pythona podzielone na kategorie. W Pythonie wszystko jest typem obiektu, nawet sam typ obiektu! Niektóre typy rozszerzeń, takie jak krotki nazwane, również mogłyby należeć do tego rysunku, ale nie spełniają formalnych kryteriów pozwalających na ich włączenie do zestawu typów podstawowych

Najistotniejszym punktem, na który należy tutaj zwrócić uwagę, jest to, że *wszystko* w systemie Python jest typem obiektu i może być przetwarzane przez programy napisane w tym języku. Na przykład możemy przekazać klasę do funkcji, przypisać ją do zmiennej czy umieścić w liście bądź słowniku itd.

Obiekty typów

Tak naprawdę nawet same typy są w Pythonie typami obiektów — typem obiektu jest obiekt typu `type` (a teraz spróbuj trzy razy szybko przeczytać na głos to stwierdzenie). A mówiąc poważnie, wywołanie wbudowanej funkcji `type(X)` zwraca typ obiektu X. W praktyce obiekty typów można wykorzystać do ręcznych porównań typów w instrukcjach `if` Pythona. Ze względu na powody wymienione w rozdziale 4. w Pythonie nie wykonuje się raczej ręcznego sprawdzania typów, gdyż ogranicza to elastyczność kodu.

Jedna uwaga dotycząca nazw typów: od Pythona w wersji 2.2 każdy typ podstawowy ma nową, wbudowaną nazwę, dodaną w celu zapewnienia możliwości dostosowywania typów do własnych potrzeb za pomocą zorientowanych obiektowo podklas: `dict` (słownik), `list` (lista), `str` (łańcuch znaków), `tuple` (krotka), `int` (liczba całkowita), `float` (liczba zmiennoprzecinkowa), `complex` (liczba zespolona), `bytes` (łańcuch bajtowy), `type` (typ), `set` (zbiór) i inne. W Pythonie 3.x te nazwy odwołują się do klas, a w wersji 2.x, ale już nie w 3.x, `file` jest także nazwą typu i synonimem funkcji `open`. Wywołania tych nazw są tak naprawdę wywołaniami konstruktora obiektu, a nie tylko funkcjami konwersji, choć można je traktować jako proste funkcje służące do realizacji podstawowych zadań.

Ponadto moduł `types` biblioteki standardowej Pythona 3.x udostępnia dodatkowe nazwy typów przeznaczone dla typów niedostępnych jako nazwy wbudowane (na przykład typu funkcji; w Pythonie 2.x, ale nie w wersji 3.x, moduł ten zawiera również synonimy dla wbudowanych nazw typów). Sprawdzanie typów można również wykonywać za pomocą funkcji `isinstance`. Na przykład wszystkie z poniższych testów są prawdziwe.

```
type([1]) == type([])          # Porównanie z typem innej listy
type([1]) == list               # Porównanie z nazwą typu listy
isinstance([1], list)          # Sprawdzenie, czy to lista, lub jej
                               # dostosowanie do własnych potrzeb
import types                   # Moduł types zawiera nazwy dla innych typów
def f(): pass
type(f) == types.FunctionType
```

Ponieważ typy mogą obecnie być wykorzystane przy tworzeniu klas podrzędnych, technika wykorzystująca funkcję `isinstance` jest mocno zalecana. Więcej informacji na temat tworzenia podklas typów wbudowanych w Pythonie 2.2 i późniejszych wersjach znajdziesz w rozdziale 32.

W rozdziale 32. sprawdzimy także, w jaki sposób funkcja `type(X)` oraz ogólnie sprawdzanie typów odnoszą się do instancji klas zdefiniowanych przez użytkownika. Mówiąc krótko, w Pythonie 3.x oraz w nowym stylu klas w Pythonie 2.x typem instancji klasy jest klasa, z której utworzona została instancja. W przypadku klasycznych klas z Pythona 2.x i wersji wcześniejszych



wszystkie instancje klas są typu „instancja” i w celu porównania ich typów musimy porównywać atrybuty `__class__` ich instancji. Ponieważ jednak nie jesteśmy jeszcze gotowi na omawianie klas, odłożymy prezentację tego zagadnienia do rozdziału 32.

Inne typy w Pythonie

Poza typami podstawowymi omawianymi w tej części książki oraz obiektami komponentów programów, takimi jak funkcje, moduły czy klasy, z którymi spotkamy się nieco później, typowa instalacja Pythona zawiera dziesiątki innych typów obiektów dostępnych jako dołączone rozszerzenia języka C lub klasy Pythona — obiekty wyrażeń regularnych, pliki DBM, widżety GUI czy gniazda sieciowe. W zależności od tego, kogo o to zapytamy, *nazwane krotki* (ang. *named tuple*), które poznaliśmy wcześniej w tym rozdziale, mogą również należeć do tej kategorii (typy `Decimal` i `Fraction` z rozdziału 5. wydają się być tutaj bardziej niejednoznaczne).

Podstawowa różnica między tymi dodatkowymi narzędziami a przedstawionymi dotychczas typami wbudowanymi polega na tym, że typy wbudowane udostępniają specjalną składnię tworzącą język dla ich obiektów (na przykład `4` dla liczb całkowitej, `[1,2]` dla listy, funkcję `open` dla plików, a także `def` oraz `lambda` dla funkcji). Inne narzędzia są zazwyczaj udostępniane przez moduły biblioteki standardowej, które przed użyciem trzeba zaimportować i które nie są zazwyczaj uważane za typy podstawowe. Na przykład, aby utworzyć obiekt wyrażenia regularnego, powinieneś zaimportować moduł `re` i wywołać metodę `re.compile()`. Wyczerpujący przewodnik po wszystkich narzędziach dostępnych dla programów napisanych w Pythonie można znaleźć w dokumentacji biblioteki standardowej tego języka.

Pułapki typów wbudowanych

To koniec naszego omówienia podstawowych typów danych. Tę część książki zamknieni przedstawieniem często spotykanych problemów, z jakimi stykają się nowi użytkownicy (a czasami nawet eksperci), wraz z ich rozwiązaniami. Część będzie powtórzeniem kwestii, z którymi już się spotkaliśmy, jednak są one na tyle ważne, że zasługują na ponowne przypomnienie.

Przypisanie tworzy referencje, nie kopie

Ponieważ jest to naprawdę kluczowa koncepcja, wspomnę o tym raz jeszcze: współdzielone odwołania (referencje) do obiektów mutowalnych mogą mieć kolosalne znaczenie dla funkcjonowania Twoich programów. Na przykład w poniższym przykładzie obiekt listy jest przypisany do zmiennej `L` i jest przywoływany zarówno ze zmiennej `L`, jak i ze środka listy przypisanej do zmiennej `M`. Modyfikacja zmiennej `L` w miejscu zmieni więc także to, do czego odwołuje się zmiana `M`.

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']                                     # Osadzenie referencji do L
>>> M
['X', [1, 2, 3], 'Y']
```

```
>>> L[1] = 0 # Zmienia również M  
>>> M  
['X', [1, 0, 3], 'Y']
```

Ten efekt nabiera znaczenia w większych programach i często współdzielone referencje są dokładnie tym, czego chcemy. Jeżeli zmieniają się w niepożądany dla Ciebie sposób, możesz uniknąć współdzielania obiektów dzięki kopowaniu ich w jawnym sposobie. W przypadku list zawsze możesz wykonać kopię najwyższego poziomu, używając wycinka z pustymi granicami lub korzystając z innych technik opisanych wcześniej:

```
>>> L = [1, 2, 3]  
>>> M = ['X', L[:], 'Y'] # Osadzenie kopii L (lub list(L)  
lub L.copy())  
>>> L[1] = 0 # Zmienia tylko L, nie M  
>>> L  
[1, 0, 3]  
>>> M  
['X', [1, 2, 3], 'Y']
```

Pamiętaj, że domyślnie granice wycinka rozciągają się od 0 do długości ciętej sekwencji. Jeżeli pominiemy obie granice, wycinek dokonuje ekstrakcji każdego elementu sekwencji, tworząc jednocześnie kopię najwyższego poziomu (czyli nowy, niewspółdzielony obiekt).

Powtórzenie dodaje jeden poziom zagębszenia

Powtórzenia w sekwencjach przypominają dodanie sekwencji do samej siebie jakąś liczbę razy. Kiedy jednak zagnieżdżone zostają zmienne sekwencje, efekt może nie zawsze być tym, czego oczekujemy. W poniższym przykładzie zmienna X zostaje przypisana do listy L powtózonej cztery razy, natomiast zmienna Y przypisana jest do listy *zawierającej* L powtózonej cztery razy.

```
>>> L = [4, 5, 6]  
>>> X = L * 4 # Jak [4, 5, 6] + [4, 5, 6] +  
...  
>>> Y = [L] * 4 # [L] + [L] + ... = [L, L, ...]  
>>> X  
[4, 5, 6, 4, 5, 6, 4, 5, 6]  
>>> Y  
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Ponieważ w drugim powtórzeniu lista L została zagnieżdzona, Y składa się z osadzonych referencji z powrotem do oryginalnej listy przypisanej do L, dlatego jest podatne na różne efekty uboczne odnotowane w poprzednim podrozdziale.

```
>>> L[1] = 0 # Ma wpływ na Y, ale nie na X  
>>> X  
[4, 5, 6, 4, 5, 6, 4, 5, 6]  
>>> Y
```

```
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

Takie sytuacje mogą się wydawać sztuczne i nieco akademickie — dopóki coś takiego nie wydarzy się niespodziewanie w Twoim kodzie! W tym scenariuszu obowiązują te same rozwiązania problemu, co wymienione w poprzednim podrozdziale, ponieważ jest to kolejny przypadek utworzenia współdzielonej referencji do mutowalnego obiektu — jeżeli nie chcesz korzystać ze współdzielonych odwołań, powinieneś zawsze tworzyć kopie obiektów:

```
>>> L = [4, 5, 6]
>>> Y = [list(L)] * 4                                # Osadzenie (współdzielonej)
kopii L
>>> L[1] = 0
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

W jeszcze bardziej subtelny sposób, choć zmienna `Y` nie dzieli już obiektu ze zmienną `L`, nadal osadza cztery odniesienia do tej samej kopii. Jeżeli musisz również unikać takiego współdzielenia, upewnij się, że każda osadzona kopia jest unikalna:

```
>>> Y[0][1] = 99                                    # Wszystkie cztery kopie są
nadaj takie same
>>> Y
[[4, 99, 6], [4, 99, 6], [4, 99, 6], [4, 99, 6]]
>>> L = [4, 5, 6]
>>> Y = [list(L) for i in range(4)]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
>>> Y[0][1] = 99
>>> Y
[[4, 99, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Jeżeli pamiętasz, że powtarzanie, konkatenacja i krojenie kopią tylko najwyższy poziom obiektów operandów, tego rodzaju przypadki mają znacznie większy sens.

Uwaga na cykliczne struktury danych

Z tą koncepcją spotkaliśmy się w poprzednim ćwiczeniu. Jeśli kolekcja obiektów zawiera referencję do samej siebie, nazywana jest *obiektem cyklicznym*. Python wyświetla [...] za każdym razem, gdy wykryje cykl w obiekcie, zamiast utknąć w nieskończonej pętli (jak to miało miejsce jeszcze jakiś czas temu):

```
>>> L = ['Graal']                                     # Dodanie referencji do tego
samego obiektu
>>> L.append(L)                                       # Utworzenie cyklu w obiekcie:
[...]
>>> L
['Graal', [...]]
```

Poza zrozumieniem, że trzy kropki w nawiasach kwadratowych oznaczają cykl w obiekcie, warto również zapoznać się z tym przypadkiem, ponieważ może on prowadzić do różnych pułapek. Struktury cykliczne mogą spowodować wpadnięcie kodu w nieskończone pętle, jeżeli nie będziemy na to przygotowani.

Na przykład niektóre programy, które przetwarzają ustrukturyzowane dane, muszą przechowywać listę, słownik lub zbiór już *odwiedzonych elementów* i sprawdzać je przed wejściem w cykl, który może spowodować niepożądaną pętlę. Więcej informacji na temat tego problemu można znaleźć w rozwiązaniach do ćwiczeń podsumowujących pierwszą część książki, zamieszczonych w dodatku D. Zwróc także uwagę na ogólne omówienie w rozdziale 19. zagadnień związanych z rekurencją, a także program *reloadall.py* w rozdziale 25. i klasę *ListTree* w rozdziale 31., gdzie znajdziesz konkretne przykłady programów, w których wykrywanie cyklu może mieć znaczenie.

Rozwiążaniem jest wiedza: nie używaj odwołać cyklicznych, jeżeli tego naprawdę nie potrzebujesz, i upewnij się, że przewidujesz i kontrolujesz je w programach, które z nich korzystają. Istnieje wiele ważnych przyczyn stosowania odwołań cyklicznych, ale jeżeli nie masz kodu obsługującego je we właściwy sposób, takie obiekty mogą zachowywać się w zaskakująco nieprzewidywalny sposób.

Typów niemutowalnych nie można modyfikować w miejscu

Na wszelki wypadek powtórzmy to jeszcze raz: obiektów niemutowalnych nie można modyfikować w miejscu. Zamiast tego powinieneś skonstruować nowy obiekt za pomocą wycinka, konkatenacji czy podobnych operacji i, jeżeli to konieczne, przypisać go z powrotem do pierwotnej referencji.

```
T = (1, 2, 3)
T[2] = 4                                # Błąd!
T = T[:2] + (4,)                          # OK: (1, 2, 4)
```

Może się to wydawać niepotrzebnym kodowaniem, jednak z drugiej strony, kiedy korzystamy z obiektów niemutowalnych, jak łańcuchy znaków i krotki, nie przytrafią się nam poprzednio opisane pułapki. Ponieważ obiektów tych nie można modyfikować w miejscu, nie są one podatne na te same efekty uboczne co listy.

Podsumowanie rozdziału

W tym rozdziale omówiliśmy ostatnie dwa spośród najważniejszych, podstawowych typów obiektów — krotkę oraz plik. Dowiedzieliśmy się, że krotki obsługują wszystkie operacje na sekwencjach, mają jedynie kilka własnych metod, nie mogą być modyfikowane w miejscu, ponieważ są niemutowalne oraz można rozszerzyć ich funkcjonalność poprzez zastosowanie krotek nazwanych. Nauczyliśmy się również, że pliki zwracane są przez wbudowaną funkcję *open* i udostępniają metody służące do odczytu oraz zapisu danych.

Sprawdziliśmy, jak można przekształcić obiekty Pythona na łańcuchy znaków służące do przechowywania danych w plikach (i odwrotnie). Przyjrzaliśmy się również modułom *pickle*, *json* i *struct* spełniającym bardziej zaawansowane role (serializacja obiektów oraz dane binarne). Wreszcie zakończyliśmy rozdział, przeglądając pewne właściwości wspólne dla wszystkich typów obiektów (na przykład współdzielone referencje) i przejrzaliśmy listę często popełnianych błędów i pułapek, na jakie jesteśmy narażeni, pracując z różnymi typami obiektów.

W kolejnej części książki zmienimy tematykę, przechodząc do zagadnień związanych ze składnią poleceń w Pythonie, czyli inaczej mówiąc do sposobów kodowania logiki przetwarzającej dane w Twoich programach. W następnych rozdziałach omówimy wszystkie podstawowe instrukcje proceduralne tego języka. Kolejny rozdział rozpoczyna tę nową część od wprowadzenia do ogólnego modelu składni Pythona, który można zastosować do wszystkich typów instrukcji. Przed przejściem dalej powinieneś jednak wykonać quiz podsumowujący rozdział, a następnie przejść przez serię ćwiczeń kończących tę część książki w celu powtórzenia najważniejszych koncepcji. Instrukcje najczęściej tworzą i przetwarzają obiekty, dlatego powinieneś się upewnić, że opanowałeś zagadnienia z dziedziny obiektów Pythona — możesz to zrobić, wykonując zamieszczone niżej ćwiczenia.

Sprawdź swoją wiedzę – quiz

1. W jaki sposób można ustalić wielkość krotki? Dlaczego narzędzie to znajduje się tam, gdzie się znajduje?
2. Napisz wyrażenie zmieniające pierwszy element krotki. Krotka `(4, 5, 6)` powinna w rezultacie stać się krotką `(1, 5, 6)`.
3. Jaka jest domyślna wartość argumentu reprezentującego tryb przetwarzania pliku w wywołaniu funkcji `open`?
4. Jaki moduł można wykorzystać do przechowywania obiektów Pythona w pliku bez ręcznego konwertowania ich na łańcuchy znaków?
5. W jaki sposób można skopiować wszystkie elementy zagnieżdżonej struktury za jednym razem?
6. Kiedy Python uznaje obiekt za prawdę?
7. Co jest Twoim celem?

Sprawdź swoją wiedzę – odpowiedzi

1. Wbudowana funkcja `len` zwraca długość (liczbę elementów) dowolnego obiektu pojemnika w Pythonie, w tym krotek. Jest to funkcja wbudowana, a nie metoda typu, ponieważ ma zastosowanie do wielu różnych typów obiektów. Ogólnie rzecz biorąc, funkcje wbudowane oraz wyrażenia mogą rozciągać się na wiele różnych typów obiektów; metody są specyficzne dla jednego typu, jednak część z nich może być dostępna dla większej ich liczby (jak na przykład `index`, działająca na listach oraz krotkach).
2. Ponieważ krotki są niemutowalne, nie można ich modyfikować w miejscu. Można jednak wygenerować nową krotkę o pożądanej wielkości. Mając krotkę `T = (4, 5, 6)`, możemy zmienić jej pierwszy element, tworząc nową krotkę z jej częścią za pomocą wycinka i konkatenacji — `T = (1,) + T[1:]`. Warto przypomnieć, że krotki jednoelementowe wymagają końcowego przecinka. Można również przekonwertować krotkę na listę, zmodyfikować ją w miejscu i przekształcić z powrotem na krotkę. Taka operacja jest jednak bardziej kosztowna i w praktyce rzadko stosowana. Jeżeli wiemy, że obiekt będzie wymagał modyfikacji w miejscu, należy od początku zastosować listę.

3. Wartością domyślną argumentu z trybem przetwarzania jest 'r' — od odczytu (ang. *read*). Aby otworzyć plik tekstowy do odczytu, wystarczy przekazać nazwę pliku zewnętrznego.
4. Do przechowania obiektów Pythona w pliku bez jawnego konwertowania ich na łańcuchy znaków można wykorzystać moduł `pickle`. Podobnym do niego modelem jest `struct`, który jednak zakłada, że dane muszą być spakowane w formacie binarnym w pliku; moduł `json` w podobny sposób konwertuje ograniczony zestaw obiektów Pythona do i z ciągów znaków zapisanych w formacie JSON.
5. Jeżeli chcemy skopiować wszystkie części zagnieźdzonej struktury X, należy zimportować moduł `copy` i wywołać `copy.deepcopy(X)`. Takie rozwiązanie rzadko jednak spotyka się w praktyce. Najczęściej chodzi nam o referencje i w większości przypadków płytkie kopie (na przykład `aList[:]`, `aDict.copy()`, `set(aSet)`) w zupełności wystarczą.
6. Obiekt uznawany jest za prawdę, kiedy jest albo liczbą inną od zera, albo niepustym obiektem kolekcji. Wbudowane słowa `True` i `False` są zdefiniowane tak, by miały to samo znaczenie, co liczby całkowite 1 i 0.
7. Poprawne odpowiedzi to m.in.: „nauczyć się Pythona”, „przejść do kolejnej części książki” albo „odnaleźć Świętego Graala”.

Sprawdź swoją wiedzę — ćwiczenia do części drugiej

W tej sesji będziemy zajmować się podstawami obiektów wbudowanych. Tak jak wcześniej, po drodze mogą pojawić się nowe koncepcje, dlatego po wykonaniu ćwiczeń (lub w ich trakcie, jeżeli sprawiają one problemy) powinieneś koniecznie przejrzeć odpowiedzi zamieszczone w sekcji „Część II — Typy i operacje” w dodatku D. Jeżeli masz mało czasu, powinieneś zacząć od ćwiczeń 10. i 11. (najbardziej praktycznych ze wszystkich), a później w wolnych chwilach kontynuować od ćwiczenia 1. Ćwiczenia obejmują fundamentalny zakres zagadnień, więc spróbuj zrobić jak najwięcej z nich; programowanie jest zajęciem praktycznym i nic nie zastąpi samodzielnego eksperymentowania z tym, co przeczytałeś w książce.

1. *Podstawy.* Poeksperymentuj z najczęściej używanymi operacjami znajdującymi się w różnych tabelach tej części książki. Na początek uruchom interpreter interaktywny Pythona, wpisz każde z poniższych wyrażeń i spróbuj wyjaśnić, co się dzieje w każdym z tych przypadków. Zwróć uwagę na to, że średnik w części przykładów został użyty do oddzielenia od siebie instrukcji, tak by udało się zmieścić kilka instrukcji w jednym wierszu. Przykładowo kod `X=1; X` przypisuje, a następnie wyświetla zmienną (więcej informacji na temat składni instrukcji znajdziesz w kolejnej części książki). Pamiętaj także, że przecinek pomiędzy wyrażeniami zazwyczaj tworzy krotkę, nawet jeżeli nie zastosowałeś nawiasów. `X, Y, Z` to krotka trójelementowa, którą Python wyświetla z użyciem nawiasów.

```
2 ** 16
2 / 5, 2 / 5.0
"mielonka" + "jajka"
S = "szynka"
"jajka " + S
```

```

S * 5
S[:0]
"zielone %s i %s" % ("jajka", S)
'zielone {0} i {1}'.format('jajka', S)
('x', )[0]
('x', 'y')[1]
L = [1,2,3] + [4,5,6]
L, L[:,], L[:0], L[-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)
{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)] = 4
list(D.keys()), list(D.values()), (1,2,3) in D
[], ["", [], (), {}, None]

```

2. *Indeksowanie i wycinki.* W sesji interaktywnej zdefiniuj listę o nazwie L, zawierającą cztery łańcuchy znaków lub liczby (na przykład L = [0, 1, 2, 3]). Następnie wykonaj kilka eksperymentów z opisanyimi niżej przypadkami granicznymi. Być może nigdy nie spotkasz ich w prawdziwych programach (a już z pewnością nie w takich dziwacznych wersjach, w jakich się tutaj pojawiają!), ale mają one na celu skłonienie Cię do myślenia o zasadach postępowania, a niektóre mogą być przydatne w innych zastosowaniach — na przykład wycinanie poza granicami może pomóc, jeżeli nie wiesz dokładnie, jakiej długości jest dana sekwencja:

- Co się stanie, kiedy spróbujemy wykorzystać indeks znajdujący się poza długością listy (jak L[4])?
- Co się stanie, kiedy spróbujemy wykonać wycinek wykraczający poza długość listy (jak L[-1000:100])?
- Jak radzi sobie Python, kiedy próbujemy dokonać ekstrakcji sekwencji w odwrotnej kolejności — gdy niższa granica jest większa od wyższej (jak L[3:1])? (Wskazówka: spróbuj przypisać coś do tego wycinka (na przykład L[3:1] = ['?']) i zobaczyć, gdzie zostanie wstawiona wartość. Czy jest to to samo zjawisko co przy próbie sporządzenia wycinka poza granicami listy?).

3. *Indeksowanie, wycinki i funkcja del.* Zdefiniuj kolejną listę czteroelementową i przypisz pustą listę do jednej z jej wartości przesunięcia (na przykład L[2] = []).

Co się stanie? Następnie przypisz pustą listę do wycinka (`L[2:3] = []`). Co stanie się teraz? Warto przypomnieć, że przypisanie do wycinka usuwa wycinek i wstawia nową wartość w miejscu, gdzie się on znajdował.

Instrukcja `del` służy do usuwania elementów o określonej wartości przesunięcia, a także kluczy, atrybutów oraz zmiennych. Można jej użyć na liście w celu usunięcia jakiegoś jej elementu (jak w `del L[0]`). Co się stanie, kiedy usuniesz cały wycinek (`del L[1:]`)? Co się stanie, kiedy przypiszesz do wycinka element niebędący sekwencją (na przykład `L[1:2] = 1`)?

4. *Przypisywanie krotek.* W sesji interaktywnej wpisz następujące wiersze kodu:

```
>>> X = 'mielonka'  
>>> Y = 'jajka'  
>>> X, Y = Y, X
```

Co stanie się z `X` i `Y` po wpisaniu tej sekwencji?

5. *Klucze słowników.* Przyjrzyj się poniższemu fragmentowi kodu:

```
>>> D = {}  
>>> D[1] = 'a'  
>>> D[2] = 'b'
```

Wiesz już, że dostęp do elementów słownika nie odbywa się po wartościach przesunięcia — co się zatem tutaj dzieje? Czy poniższy kod choć trochę to wyjaśnia? (Wskazówka: do jakiej kategorii typów należą łańcuchy znaków, liczby całkowite oraz krotki?)

```
>>> D[(1, 2, 3)] = 'c'  
>>> D  
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Indeksowanie słowników.* Utwórz słownik `D` z trzema wpisami dla kluczy '`a`', '`b`' oraz '`c`'. Co się stanie, kiedy spróbujesz uzyskać dostęp do nieistniejącego klucza (`D['d']`)? Co zrobi Python, kiedy spróbujesz przypisać coś do nieistniejącego klucza '`d`' (na przykład `D['d'] = 'mielonka'`)? Jak można to porównać z przypisaniami i referencjami poza granicami listy? Czy wygląda to na regułę dotyczącą nazw zmiennych?

7. *Operacje uniwersalne.* Wykorzystaj sesję interaktywną do uzyskania odpowiedzi na poniższe pytania.

- Co się stanie, kiedy spróbujesz wykorzystać operator `+` na różnych (mieszanych) typach danych (na przykład łańcuchu znaków i liście, liście i krotce)?
- Czy `+` działa, kiedy jeden z argumentów jest słownikiem?
- Czy metoda `append` działa zarówno dla list, jak i łańcuchów znaków? Czy można użyć metody `keys` na listach? Wskazówka: co metoda `append` zakłada na temat obiektu będącego jej podmiotem?
- Wreszcie, jaki typ obiektu otrzymasz, kiedy sporządzisz wycinek lub wykonasz konkatenację dwóch list lub dwóch łańcuchów znaków?

8. *Indeksowanie łańcuchów znaków.* Zdefiniuj łańcuch znaków S zawierający cztery znaki: S = "jajo". Później wpisz wyrażenie S[0][0][0][0]. Czy masz jakiś pomysł, co może się teraz wydarzyć? Wskazówka: warto przypomnieć, że łańcuch znaków jest kolekcją znaków, a znaki w Pythonie to łańcuchy jednoznakowe. Czy takie wyrażenie indeksujące nadal będzie działać, kiedy zastosujemy je do listy, takiej jak ['j', 'a', 'j', 'o']? Dlaczego?
9. *Typy niemutowalne.* Ponownie zdefiniuj łańcuch znaków S zawierający cztery znaki: S = "jajo". Napisz instrukcję przypisania zmieniającą ten łańcuch znaków na "jaja", używając do tego wyłącznie wycinka oraz konkatenacji. Czy tę samą operację można wykonać z użyciem samego indeksowania oraz konkatenacji? A przypisania do indeksu?
10. *Zagnieżdżanie.* Utwórz strukturę danych reprezentującą dane osobowe — imiona i nazwisko, wiek, zawód, adres, adres e-mail i numer telefonu. Strukturę tę możesz zbudować z użyciem dowolnej kombinacji wbudowanych typów obiektów (list, krotek, słowników, łańcuchów znaków, liczb). Następnie uzyskaj dostęp do poszczególnych komponentów struktury danych za pomocą indeksowania. Czy pewne struktury sprawdzają się w tym obiekcie lepiej od innych?
11. *Pliki.* Napisz skrypt tworzący nowy plik wyjściowy o nazwie *myfile.txt* i zapisujący do niego łańcuch znaków "Witaj, wspaniały świecie!". Później napisz kolejny skrypt otwierający plik *myfile.txt*, odczytujący i wyświetlający jego zawartość. Oba skrypty wykonaj z poziomu systemowego wiersza poleceń. Czy nowy plik pojawi się w katalogu, w którym wykonałeś skrypty? Co się stanie, jeżeli do nazwy pliku przekazanej do metody *open* dodamy inną ścieżkę do katalogu? Uwaga: metody *write* nie dodają znaków nowego wiersza do zapisywany tekstu. Jeżeli chcesz w pełni zakończyć wiersz w pliku, musisz dodać \n na końcu łańcucha znaków.

[1] Jest jeszcze bardziej subtelna przyczyna — przecinek jest rodzajem operatora o najniższym priorytecie, ale tylko w kontekstach, w których nie ma innego znaczenia. W takich kontekstach to przecinek, a nie nawiasy, tworzy krotki; powoduje to, że nawiasy stają się opcjonalne, ale może również prowadzić do dziwnych, nieoczekiwanych błędów składniowych, jeżeli nawiasy zostaną pominięte w niewłaściwym momencie.

[2] *Big-endian* to forma zapisu danych, w której najbardziej znaczący bajt umieszczony jest jako pierwszy — przyp. tłum.

Część III Instrukcje i składnia

Rozdział 10. Wprowadzenie do instrukcji Pythona

Ponieważ znasz już podstawowe typy obiektów Pythona, niniejszy rozdział rozpoczniemy od omówienia podstawowych form instrukcji tego języka. Tak jak w poprzedniej części, zaczniemy od ogólnego wprowadzenia do składni instrukcji. W kolejnych rozdziałach znajdą się bardziej szczegółowe informacje dotyczące poszczególnych instrukcji.

Mówiąc ogólnie, *instrukcje* (ang. *statements*) to coś, co piszemy w celu przekazania Pythonowi tego, co mają robić nasze programy. Jeżeli, tak jak to sugerowaliśmy w rozdziale 4., program „coś robi”, to właśnie instrukcje pozwalają określić, co to jest. Mniej formalnie mówiąc, Python jest językiem proceduralnym, opartym na instrukcjach; łącząc instrukcje w odpowiednie bloki, tworzymy *procedury* wykonywane przez Pythona w celu spełnienia zadania programu.

Raz jeszcze o hierarchii pojęciowej języka Python

Innym sposobem zrozumienia roli instrukcji jest powrócenie do hierarchii wprowadzonej w rozdziale 4., gdzie omawialiśmy obiekty wbudowane wraz z wyrażeniami służącymi do ich przetwarzania. Niniejszy rozdział stanowi przejście o jeden poziom w górę hierarchii struktury programu:

1. Programy składają się z modułów.
2. Moduły składają się z instrukcji.
3. *Instrukcje zawierają wyrażenia.*
4. Wyrażenia tworzą i przetwarzają obiekty.

Ogólnie rzecz biorąc, programy napisane w Pythonie składają się z instrukcji i wyrażeń. Wyrażenia przetwarzają obiekty i są osadzone w instrukcjach. Instrukcje tworzą *logikę* działania programu — wykorzystują i kierują wyrażenia do przetwarzania obiektów omawianych w poprzednich rozdziałach. Ponadto to właśnie w instrukcjach obiekty zaczynają istnieć (na przykład w wyrażeniach wewnątrz instrukcji przypisania), a niektóre instrukcje tworzą zupełnie nowe rodzaje obiektów (na przykład funkcje i klasy). Instrukcje zawsze istnieją w modułach, które z kolei same są zarządzane za pomocą instrukcji.

Instrukcje Pythona

W tabeli 10.1 zaprezentowano zbiór instrukcji Pythona. Każda instrukcja w Pythonie ma swoje konkretne przeznaczenie i swoją własną *składnię* (zbiór reguł określających jej strukturę), ale jak się niebawem przekonasz, wiele poleceń posiada wspólne wzorce składniowe, a role

niektórych instrukcji są do siebie podobne bądź nawet się w dużej mierze pokrywają. Tabela 10.1 zawiera również przykłady poszczególnych instrukcji, kodowanych zgodnie z ich regułami składniowymi. W programach możesz używać instrukcji do wykonywania akcji, powtarzania zadań, dokonywania wyborów, budowania większych struktur programów i tak dalej.

W tej części książki omówimy polecenia z tabeli od jej początku aż do instrukcji `break` i `continue`. Niektóre instrukcje z tabeli zostały już nieformalnie wprowadzone wcześniej. W tej części książki uzupełnimy pominięte szczegóły, wprowadzimy pozostałą część zbioru instrukcji proceduralnych Pythona i omówimy ogólny model składni. Instrukcje z drugiej części tabeli 10.1, dotyczące większych części programów — funkcji, klas, modułów oraz wyjątków — prowadzą do zadań programistycznych, dlatego zasługują na poświęcenie im osobnych części. Instrukcje bardziej wyspecjalizowane (jak `del`, usuwająca różne komponenty) omówione są w dalszej części książki lub w dokumentacji biblioteki standardowej Pythona.

Technicznie rzecz biorąc, w tabeli 10.1 znajdują się instrukcje Pythona w wersji 3.x — choć jej zawartość może spełniać rolę szybkiego przeglądu polecen Pythona, to jednak z oczywistych względów nie jest kompletna. Poniżej zamieszczamy kilka uwag na temat jej zawartości:

- Instrukcje przypisania mogą mieć różne formy składni, opisane w rozdziale 11. — prostą, sekwencyjną, rozszerzoną i inne.
- `print` w wersji 3.x nie jest ani słowem zarezerwowanym, ani instrukcją; jest to wywołanie funkcji wbudowanej. Ponieważ jednak prawie zawsze wykonywane jest w postaci instrukcji wyrażenia (czyli w oddzielnym wierszu), zazwyczaj traktuje się je jako typ instrukcji. Operacje z użyciem funkcji `print` zostaną omówione w rozdziale 11.
- Począwszy od wersji 2.5, `yield` jest tak naprawdę wyrażeniem, a nie instrukcją. Tak jak `print`, najczęściej wykorzystywane jest w oddzielnym wierszu, dlatego uwzględnione zostało w tej tabeli, jednak jak zobaczymy w rozdziale 20., w skryptach wynik działania tej instrukcji jest czasami przypisywany do zmiennej lub wykorzystywany w inny sposób. Jako wyrażenie `yield` jest także słowem zarezerwowanym — odwrotnie niż `print`.

Tabela 10.1. Instrukcje języka Python

Instrukcja	Rola	Przykład
Przypisanie	Tworzenie referencji	<code>a, b = 'dobry', 'zły'</code>
Wywołania i inne wyrażenia	Wywoływanie funkcji	<code>log.write("mielonka, szynka")</code>
Wywołania <code>print</code>	Wyświetlanie obiektów	<code>print('The Killer', joke)</code>
<code>if/elif/else</code>	Wybór działania	<code>if "python" in text: print(text)</code>
<code>for/else</code>	Iteracje	<code>for x in mylist: print(x)</code>
<code>while/else</code>	Ogólne pętle	<code>while X > Y: print('witaj')</code>
<code>pass</code>	Pusta instrukcja zastępcza	<code>while True: pass</code>
<code>break</code>	Wyjście z pętli	<code>while True: if exittest(): break</code>
<code>continue</code>	Kontynuacja pętli	<code>while True:</code>

		<code>if skiptest(): continue</code>
<code>def</code>	Funkcje i metody	<code>def f(a, b, c=1, *d): print(a+b+c+d[0])</code>
<code>return</code>	Wynik funkcji	<code>def f(a, b, c=1, *d): return a+b+c+d[0]</code>
<code>yield</code>	Funkcje generatora	<code>def gen(n): for i in n: yield i*2</code>
<code>global</code>	Przestrzenie nazw	<code>x = 'stary' def function(): global x, y; x = 'nowy'</code>
<code>nonlocal</code>	Przestrzenie nazw (3.x)	<code>def outer(): x = 'stary' def function(): nonlocal x; x = 'nowy'</code>
<code>import</code>	Dostęp do modułów	<code>import sys</code>
<code>from</code>	Dostęp do atrybutów	<code>from sys import stdin</code>
<code>class</code>	Budowanie obiektów	<code>class Subclass(Superclass): staticData = [] def method(self): pass</code>
<code>try/except/finally</code>	Przechwytywanie wyjątków	<code>try: action() except: print('Błąd w akcji')</code>
<code>raise</code>	Zgłaszanie wyjątków	<code>raise EndSearch(location)</code>
<code>assert</code>	Sprawdzanie w debugowaniu	<code>assert X > Y, 'X jest za małe'</code>
<code>with/as</code>	Menedżery kontekstu (3.x, 2.6+)	<code>with open('data') as myfile: process(myfile)</code>
<code>del</code>	Usuwanie referencji	<code>del dane[k] del dane[i:j] del obiekt.atrybut del zmienna</code>

Większość powyższej tabeli ma także zastosowanie do Pythona 2.x, choć z pewnymi wyjątkami — jeżeli używasz Pythona 2.x, powinieneś zapoznać się z kilkoma uwagami dotyczącymi tej wersji Pythona:

- W wersji 2.x instrukcja `nonlocal` nie jest dostępna. Jak zobaczymy w rozdziale 17., istnieją alternatywne sposoby pozwalające uzyskać funkcjonalność udostępnianą przez tę instrukcję.
- W wersji 2.x `print` jest instrukcją, a nie wywołaniem wbudowanej funkcji, ze składnią omówioną w rozdziale 11.

- W wersji 2.x wbudowana funkcja `exec` jest instrukcją o zdefiniowanej składni. Ponieważ jednak obsługuje ona nawiasy, można używać wywołania z wersji 3.x także w Pythonie 2.x.
- W wersji 2.5 instrukcje `try/except` oraz `try/finally` zostały połączone. Formalnie były one oddzielnymi instrukcjami, jednak teraz możliwe jest użycie zarówno `except`, jak i `finally` w tej samej instrukcji `try`.
- W wersji 2.5 `with/as` jest opcjonalnym rozszerzeniem i nie jest dostępne, jeżeli niełączymy go w jawnym sposób, wykonując instrukcję `from __future__ import with_statement` (więcej informacji na ten temat znajduje się rozdziale 34.).

Historia dwóch if

Zanim zagłębimy się w szczegóły którejś z instrukcji z tabeli 10.1, chciałbym zacząć nasze omawianie składni instrukcji od pokazania, czego *nie* będziemy wpisywać do kodu Pythona, tak by można było dokonać porównania tego języka z innymi modelami składni, z jakimi można się spotkać.

Rozważmy poniższą instrukcję `if` zakodowaną w języku podobnym do C.

```
if (x > y) {
    x = 1;
    y = 2;
}
```

Może to być instrukcja w języku C, C++, Java, JavaScript lub podobnego. Teraz przyjrzyjmy się odpowiadającej jej instrukcji z Pythona.

```
if x > y:
    x = 1
    y = 2
```

Pierwszą rzeczą, jaką łatwo zauważyc, jest to, że instrukcja napisana w Pythonie jest mniej, nazywamy to, zaśmiecona — to znaczy jest w niej mniej elementów składniowych. Jest to celowe — Python jest językiem skryptowym, zatem jednym z jego celów jest ułatwienie życia programistom poprzez pisanie mniejszej ilości kodu.

Co więcej, kiedy porównamy oba modele składni, okaże się, że Python dodaje jeden dodatkowy element, a trzy elementy obecne w językach podobnych do C w Pythonie są nieobecne.

Co dodaje Python

Tym jednym dodatkowym elementem składniowym jest w Pythonie znak dwukropka (`:`). Wszystkie *instrukcje złożone* w Pythonie (czyli instrukcje z zagnieżdżonymi kolejnymi instrukcjami) pisze się zgodnie z jednym wzorcem — z nagłówkiem zakończonym dwukropkiem, po którym następuje zagnieżdżony blok kodu wcięty w stosunku do wiersza nagłówka.

Wiersz nagłówka:

Zagnieżdżony blok instrukcji

Dwukropki jest wymagany i pominięcie go jest chyba najczęściej popełnianym przez początkujących programistów Pythona błędem — na swoich szkoleniach i kursach widziałem go tysiące razy. Każda osoba zaczynająca swoją przygodę z Pythonem szybko zapomina o znaku

dwukropka. Większość edytorów do Pythona sprawia, że błąd ten jest łatwo zauważyc, a wpisywanie dwukropka w końcu staje się nieświadomym nawykem (do tego stopnia, że można odruchowo zacząć wpisywać dwukropki do kodu napisanego w językach podobnych do C, generując tym samym wiele zabawnych komunikatów o błędach ze strony kompilatora takiego języka).

Co usuwa Python

Wprawdzie Python wymaga dodatkowego znaku dwukropka, za to programiści w językach podobnych do C muszą uwzględnić trzy rzeczy, których generalnie nie trzeba robić w Pythonie.

Nawiąsy są opcjonalne

Pierwszym z nich są nawiasy wokół testów znajdujących się na początku instrukcji.

```
if (x < y)
```

Nawiąsy wymagane są przez składnię wielu języków podobnych do C. W Pythonie tak jednak nie jest — nawiasy możemy pominąć, a instrukcja nadal będzie działać.

```
if x < y
```

Z technicznego punktu widzenia, ponieważ każde wyrażenie można umieścić w nawiasach, wstawienie ich tutaj nie zaszkodzi i nie będą one potraktowane jako błąd.

Nie należy tego jednak robić — to niepotrzebnie zwiększa zużycie klawiatury, a na dodatek zdradza całemu światu, że jesteśmy byli programistami języka C, którzy nadal uczą się Pythona (sam takim kiedyś byłem). „Pythonowy” sposób polega na całkowitym pomijaniu nawiasów w tego rodzaju instrukcjach.

Koniec wiersza jest końcem instrukcji

Drugim, bardziej znaczącym elementem składni, którego nie znajdziemy w Pythonie, jest znak średnika (;). W Pythonie nie trzeba kończyć instrukcji za pomocą średników, tak jak robi się to w językach podobnych do C.

```
x = 1;
```

W Pythonie istnieje ogólna reguła mówiąca, że koniec wiersza jest automatycznie końcem instrukcji znajdującej się w tym wierszu. Innymi słowy, można opuścić średniki, a instrukcja będzie działała tak samo.

```
x = 1
```

Istnieje kilka obejść tej reguły, o czym przekonamy się za chwilę (na przykład opakowanie fragmentów kodu w strukturę ujętą w nawiasy pozwala na kontynuację polecenia w kolejnych wierszach). Generalnie jednak w większości przypadków w Pythonie umieszczamy po jednej instrukcji w wierszu i średniki nie są wymagane.

Również tutaj osoby tęskniące za programowaniem w języku C (o ile to w ogóle możliwe...) mogą nadal używać średników na końcu każdej instrukcji — sam język nam na to pozwala, ponieważ średnik spełnia również rolę separatora poleceń, jeżeli kilka instrukcji zostanie umieszczone w jednym wierszu.

Jednak podobnie jak poprzednio, *nie należy tego robić* (naprawdę!) — kolejny raz zdradza to, że nadal jesteś programistą wychowanym na języku podobnym do C, który jeszcze nie przestawił się na kodowanie w Pythonie. Programując w Pythonie, po prostu nie stosujemy średników. Sądząc po studentach, których spotykam na moich kursach, dla wielu doświadczonych programistów przełamanie takich głęboko zakorzenionych nawyków wydaje się być niezwykle trudne, ale da się to zrobić — w takiej roli średniki w Pythonie są tylko niepotrzebnym szumem.

Koniec wcięcia to koniec bloku

Trzecim i ostatnim komponentem składniowym nieobecnym w Pythonie, i chyba najbardziej niezwykłym dla byłych programistów języka C (dopóki nie przywykną do tego po kilkunastu minutach programowania i nie uświadomią sobie, że to właściwie zaleta!), jest to, że w kodzie nie wpisujemy niczego, co jawnie oznaczałoby początek i koniec zagnieżdzonego bloku kodu. Nie musimy wpisywać słów kluczowych `begin/end`, `then/endif` czy umieszczać wokół kodu nawiasów klamrowych, tak jak robi się to w językach podobnych do C:

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

Zamiast tego w Pythonie w spójny sposób wcina się wszystkie instrukcje w danym bloku zagnieżdżonym o tę samą odległość w prawo. Python wykorzystuje fizyczne rozmiary wcięć fragmentów kodu do ustalenia, gdzie blok się zaczyna i gdzie się kończy.

```
if x > y:  
    x = 1  
    y = 2
```

Przez *wcięcia* rozumiemy puste białe znaki znajdujące się po lewej stronie obu zagnieżdżonych instrukcji. Pythona nie interesuje sposób tworzenia wcięć (można korzystać ze spacji lub tabulatorów) ani też ich *ilość* (można użyć dowolnej liczby spacji lub tabulatorów). Wcięcie jednego bloku zagnieżdżonego może tak naprawdę być zupełnie inne od wcięcia innego bloku. Reguła składni mówi jedynie, że w jednym bloku zagnieżdżonym wszystkie instrukcje muszą być wcięte na tę samą odległość w prawo. Jeżeli tak nie jest, otrzymamy błąd składni i kod nie zostanie wykonany, dopóki nie ustawiemy rozmiarów wcięć w spójny sposób.

Skąd bierze się składnia z użyciem wcięć

Reguła wcięć może programistom języków podobnych do C na pierwszy rzut oka wydać się niezwykła, jednak jest ona celową cechą Pythona oraz jednym ze sposobów, w jaki Python wymusza na programistach tworzenie jednolitego, regularnego i czytelnego kodu. Oznacza to, że kod musi być wyrównany w pionie, w kolumnach, zgodnie ze swoją strukturą logiczną. Rezultat jest taki, że kod staje się bardziej spójny i czytelny (w przeciwieństwie do kodu napisanego w językach podobnych do C).

Mówiąc inaczej, wyrównanie kodu zgodnie z jego strukturą logiczną jest podstawowym narzędziem uczynienia go czytelnym i tym samym łatwym w ponownym użyciu oraz późniejszym utrzymywaniu — zarówno przez nas samych, jak i przez inne osoby. Nawet osoby, które po skończeniu lektury niniejszej książki nigdy nie będą używać Pythona, powinny nabrać nawyku wyrównywania kodu w celu zwiększenia jego czytelności w dowolnym języku o strukturze blokowej. Python wymusza to, gdyż jest to częścią jego składni, jednak ma to znaczenie w każdym języku programowania i ogromny wpływ na użyteczność naszego kodu.

Doświadczenia każdej osoby mogą być różne, jednak kiedy ja byłem pełnoetatowym programistą, byłem zatrudniony przede wszystkim przy pracy z dużymi, starymi programami w języku C++, tworzonymi przez długie lata przez wiele różnych osób. Co było nie do uniknięcia, prawie każdy programista miał swój własny styl tworzenia wcięć kodu. Często proszono mnie na przykład o zmianę pętli `while` zakodowanej w języku C, która rozpoczęła się w następujący sposób:

```
while (x > 0) {
```

Zanim jeszcze przejdziemy do samego tworzenia wcięć, warto zauważać, że istnieją trzy czy cztery sposoby układania nawiasów klamrowych w językach podobnych do C. Wiele organizacji prowadzi niemal polityczne debaty i tworzy podręczniki opisujące standardy, które powinny sobie z tym radzić (co wydaje się nieco przesadne, zważając na to, że trudno uznać to za problem programistyczny). Ignorując te spory, przejdźmy do scenariusza, z jakim często spotykałem się w kodzie napisanym w języku C++. Pierwsza osoba pracująca nad tym kodem wcinała pętlę o cztery spacje.

```
while (x > 0) {  
    -----;  
    -----;
```

Ta osoba w końcu awansowała, zajęła się zarządzaniem, a jej miejsce zajął ktoś, kto wymuszał wcięcia jeszcze bardziej przesunięte w prawo.

```
while (x > 0) {  
    -----;  
    -----;  
    -----;  
    -----;  
    -----;
```

I ta osoba w pewnym momencie zmieniła pracę (kończąc rządy „wcięciowego” terroru), a jej zadania przejął ktoś, kto wolał mniejsze wcięcia.

```
while (x > 0) {  
    -----;  
    -----;  
    -----;  
    -----;  
    -----;  
    -----;  
    -----;  
    -----;  
}
```

I tak w nieskończoność. Blok ten kończy się nawiasem klamrowym (}), co oczywiście sprawia, że jest on kodem ustrukturyzowanym blokowo (przynajmniej teoretycznie). W każdym języku ustrukturyzowanym blokowo (w Pythonie i innych), jeżeli zagnieżdżone bloki nie są wcięte w spójny sposób, stają się trudne do interpretacji, modyfikacji czy ponownego użycia, ponieważ kod wizualnie nie odzwierciedla już swojego znaczenia logicznego. *Czytelność ma duże znaczenie* i wcięcia są jej istotnym komponentem.

Poniżej znajduje się kolejny przykład, z którym mogłeś się zetknąć, jeżeli programowałeś kiedyś w języku podobnym do C. Rozważmy poniższą instrukcję języka C:

```
if (x)  
    if (y)  
        instrukcja1;  
    else  
        instrukcja2;
```

Z którym `if` powiązane jest `else`? Co może być zaskoczeniem, `else` jest dopełnieniem zagnieździonej instrukcji `if (y)`, choć wizualnie wydaje się przynależeć do zewnętrznej instrukcji `if (x)`. To klasyczna pułapka języka C, która może prowadzić do całkowicie niepoprawnej interpretacji kodu przez użytkownika i jego modyfikacji w niepoprawny sposób, co może nie zostać wykryte do momentu, gdy marsjański łazik rozbije się na wielkiej skale!

Takie coś nie może się jednak zdarzyć w Pythonie. Ponieważ wcięcia kodu są znaczące, wygląd kodu odzwierciedla w pewnym sensie jego działanie. Rozważmy odpowiednik powyższego kodu napisany w Pythonie:

```
if x:  
    if y:  
        instrukcja1  
    else:  
        instrukcja2
```

W tym przykładzie `if`, z którym `else` wyrównane jest w pionie, to `if`, z którym `else` jest powiązane logicznie (jest to zewnętrzne `if x`). W pewnym sensie Python jest językiem typu WYSIWYG (ang. *What You See Is What You Get* — to co widzisz, jest tym, co otrzymujesz), ponieważ kod wykonywany jest tak, jak wygląda, bez względu na to, kto go napisał.

Jeżeli te argumenty nie były w stanie przekonać kogoś o wyższości składni Pythona, podam jeszcze jedną anegdotę. Na początku mojej kariery zawodowej pracowałem w firmie rozwijającej oprogramowanie w języku C, w którym spójne wcięcia nie były wymagane. Mimo to, kiedy pod koniec dnia przesyśaliśmy kod do systemu kontroli wersji, wykorzystywano zautomatyzowany skrypt analizujący wcięcia w kodzie. Jeżeli skrypt zauważał, że nie wcinamy kodu w sposób spójny, następnego dnia czekał na nas e-mail w tej sprawie, który trafiał również do naszych szefów.

Dlaczego o tym piszę? Nawet jeżeli język programowania tego nie wymaga, dobrzy programiści wiedzą, że spójne wcięcia kodu mają ogromny wpływ na jego czytelność i jakość. To, że Python przenosi tę kwestię na poziom składni, przez większość osób uznawane jest za wielką zaletę.

Wreszcie należy także pamiętać, że prawie każdy edytor tekstu dla programistów ma wbudowaną obsługę modelu składni Pythona. W IDLE wiersze kodu są wcinane automatycznie, kiedy tworzymy blok zagnieżdzony. Naciśnięcie przycisku *Backspace* powraca o jeden poziom wcięcia i można również ustawić, jak daleko do prawej strony IDLE wcina instrukcje zagnieżdzonego bloku. Nie ma uniwersalnego standardu określającego sposób wcinania kodu. Często stosuje się cztery spacje lub jeden tabulator na poziom, jednak w efekcie końcowym to każdy z nas decyduje, w jaki sposób i na jaką odległość wcinać kod (o ile nie pracujesz nad projektem, w którym wielkość wcięć została z takich i czy innych powodów ustalonyzowana). Dla bloków bardziej zagnieżdzonych odległość ta może być większa, dla bloków bliższych zewnętrznemu — mniejsza.

Jako ogólną zasadę należy przyjąć niemieszanie tabulatorów i spacji w tym samym bloku w kodzie Pythona, o ile takie formatowanie nie jest stosowane w spójny sposób. W jednym bloku powinny być używane albo tabulatory, albo spacje, ale nie jedno i drugie rozwiązywanie jednocześnie (jak zobaczymy w rozdziale 12., Python 3.x zwraca teraz błąd w przypadku niekonsekwentnego wykorzystywania tabulatorów i spacji). Spacji i tabulatorów nie powinno się tak naprawdę mieszać w żadnym ustrukturyzowanym języku programowania — taki kod powoduje znaczne problemy z czytelnością, jeżeli kolejny programista ma w swoim edytorze ustalony inny sposób wyświetlania tabulatorów. Programistom języków takich jak C może to ujście na sucho, choć nie powinno tak być — w ten sposób powstaje ogromny bałagan.

Muszę zatem powtórzyć: bez względu na to, w jakim języku programujesz, z uwagi na czytelność kodu powinieneś tworzyć wcięcia w konsekwentny i spójny sposób. Jeżeli ktoś się tego nie nauczył na początku swej kariery programistycznej, to znaczy, że jego nauczyciele

wyśliadczyli mu niedźwiedzią przysługę. Większość programistów — a już zwłaszcza ci, którzy muszą czytać kod napisany przez inne osoby — uznają za zaletę to, że Python podnosi tworzenie wcięć na poziom składni. Co więcej, wstawianie tabulatorów zamiast nawiasów klamrowych nie jest w praktyce trudniejsze dla narzędzi zwracających kod w Pythonie. Generalnie wystarczy robić to samo co w językach podobnych do C — należy się tylko pozbyć nawiasów klamrowych, a kod będzie spełniał reguły składni Pythona.

Kilka przypadków specjalnych

Jak wspominaliśmy wcześniej, w modelu składni Pythona:

- koniec wiersza kończy instrukcję znajdująca się w tym wierszu (bez konieczności używania średników),
- instrukcje zagnieżdżone są łączone w bloki i wiązane ze sobą za pomocą fizycznego wcięcia (bez konieczności używania nawiasów klamrowych).

Te dwie proste reguły odnoszą się niemal do całego kodu napisanego w Pythonie, z jakim możesz się spotkać w praktyce. Python udostępnia również kilka reguł specjalnego przeznaczenia, które pozwalają na dostosowanie instrukcji i zagnieżdżonych bloków instrukcji do własnych potrzeb. Nie są one jednak wymagane i powinny być używane oszczędnie, ale programiści uznali je za przydatne w praktyce.

Przypadki specjalne dla reguły o końcu wiersza

Choć instrukcje normalnie pojawiają się po jednej na wiersz, można również umieścić w wierszu kodu Pythona więcej niż jedną instrukcję, rozdzielając je od siebie średnikami.

```
a = 1; b = 2; print(a + b) # Trzy instrukcje w wierszu
```

To jedyne miejsce, w którym w Pythonie wymagane są średniki — jako *separatory instrukcji*. Działa to jednak tylko wtedy, gdy połączone w ten sposób instrukcje nie są instrukcjami złożonymi. Innymi słowy, można połączyć ze sobą jedynie proste instrukcje, takie jak przypisania, wyświetlanie za pomocą `print` czy wywołania funkcji. Instrukcje złożone, takie jak testy `if` czy pętle `while`, nadal muszą pojawiać się w osobnych wierszach (w przeciwnym razie teoretycznie w jednym wierszu mógłbyś umieścić cały program, co z pewnością nie przysporzyłoby Ci popularności wśród współpracowników).

Druga reguła specjalna dla instrukcji jest w zasadzie odwróceniem tej pierwszej: jedna instrukcja może rozciągać się na kilka wierszy. Aby to zadziałało, wystarczy umieścić część instrukcji w parze nawiasów — zwykłych `()`, kwadratowych `[]` lub klamrowych `{}`. Kod umieszczony w tych konstrukcjach może znajdować się w kilku wierszach. Instrukcja nie kończy się, dopóki Python nie dojdzie do wiersza zawierającego nawias zamykający. Przykładem może być rozciągający się na kilka wierszy literał listy.

```
mlist = [111,  
          222,  
          333]
```

Ponieważ kod umieszczony jest w parze nawiasów kwadratowych, Python po prostu przechodzi do kolejnego wiersza aż do momentu napotkania nawiasu zamykającego. Nawiasy klamrowe otaczające słowniki (a także literaly zbiorów oraz słowniki i listy składane w wersjach 3.x i 2.7) także mogą się rozciągać na kilka wierszy, natomiast zwykłe nawiasy mogą mieć krotki, wywołania funkcji i wyrażenia. Wcięcia wierszy kontynuacji nie mają znaczenia, choć zdrowy rozsądek nakazuje jakość wyrównać ze sobą kolejne wiersze dla celów czytelności.

Nawiasy są wszechstronnym narzędziem. Ponieważ można w nich umieścić dowolne wyrażenie, samo wstawienie lewego nawiasu pozwala na przejście do kolejnego wiersza i kontynuowanie

instrukcji tam.

```
X = (A + B +  
      C + D)
```

Ta technika działa zresztą również w przypadku instrukcji złożonych. Kiedy tylko potrzebujemy zakodować duże wyrażenie, wystarczy opakować je w nawiasy, by móc je kontynuować w kolejnym wierszu.

```
if (A == 1 and  
    B == 2 and  
    C == 3):  
    print('mielonka' * 3)
```

Starsza reguła pozwala również na kontynuację w następnym wierszu, kiedy poprzedni kończy się ukośnikiem lewym.

```
X = A + B + \  
     C + D          # starsze rozwiązanie alternatywne, które  
     jest podatne na błędy
```

Ta alternatywna technika jest jednak przestarzała i dzisiaj jest źle oceniana, ponieważ trudno jest zauważyc i utrzymać znaki lewego ukośnika. Jest również dość krucha i podatna na błędy — po lewym ukośniku nie może być spacji, a przypadkowe pominięcie może mieć nieoczekiwane skutki, jeżeli następny wiersz zostanie uznany za nową instrukcję (w tym przykładzie „C + D” jest poprawną instrukcją samo w sobie, nawet jeżeli nie jest wcięte). Ta reguła jest także kolejnym powrotem do języka C, gdzie jest powszechnie używana w makrach „#define” — pamiętaj: gdy jesteś w krainie Pythona, postępuj tak jak „pythoniści”, a nie jak programiści języka C.

Przypadki specjalne dla reguły o wcięciach bloków

Jak wspomniano wcześniej, instrukcje w zagnieźdzonym bloku kodu są zazwyczaj wiązane ze sobą dzięki wcinaniu na tę samą odległość w prawą stronę. W specjalnym przypadku ciało instrukcji złożonej może pojawić się w tym samym wierszu co jej nagłówek, po znaku dwukropka.

```
if x > y: print(x)
```

Pozwala to na kodowanie jednowierszowych instrukcji if czy pętli. Tutaj jednak zadziała to tylko wtedy, gdy ciało instrukcji złożonej nie zawiera żadnych instrukcji złożonych. Mogą się tam znajdować jedynie proste instrukcje — przypisania, instrukcje print, wywołania funkcji i tym podobne. Większe instrukcje nadal muszą być umieszczane w osobnych wierszach. Dodatkowe części instrukcji złożonych (na przykład część else z if, z którą spotkamy się później) również muszą znajdować się w osobnych wierszach. Ciało instrukcji może składać się z kilku prostych instrukcji rozdzielonych średnikami, jednak takie rozwiązanie zazwyczaj nie jest pochwalane.

Ogólnie rzecz biorąc — mimo że nie zawsze jest to wymagane — jeżeli będziemy umieszczać każdą instrukcję w osobnym wierszu i zawsze wcinać zagnieździone bloki, nasz kod będzie łatwiejszy do odczytania i późniejszej modyfikacji. Co więcej, niektóre narzędzia służące do profilowania kodu i testów pokrycia mogą nie być w stanie rozróżnić kilku instrukcji umieszczonych w jednym wierszu czy też oddzielić od siebie nagłówka i ciała jednowierszowej instrukcji złożonej. W Pythonie prawie zawsze prostota daje najlepsze rezultaty. Opisane wyżej wyjątki możesz wykorzystać do napisania w języku Python specjalnie zagmatwanego, trudnego do odczytania kodu, ale wymaga to sporego wysiłku i prawdopodobnie są lepsze sposoby na spędzanie wolnego czasu.

Aby jednak zobaczyć w działaniu najważniejszy i powszechnie stosowany wyjątek od jednej z tych reguł (użycie instrukcji `if` oraz `break` w jednym wierszu do wyjścia z pętli) i wprowadzić więcej składni Pythona, przejdziemy do następnej sekcji i zaczniemy pisać prawdziwy kod.

Szybki przykład – interaktywne pętle

Wszystkie te reguły składni zobaczymy w działaniu, kiedy w kolejnych rozdziałach będziemy omawiać określone instrukcje złożone Pythona. Działają one w ten sam sposób w całym języku. Na początek zajmiemy się krótkim, realistycznym przykładem demonstrującym sposób łączenia składni i zagnieżdżania instrukcji, a także wprowadzającym przy okazji kilka instrukcji.

Prosta pętla interaktywna

Przypuśćmy, że zostałeś poproszony o napisanie w Pythonie programu wchodzącego w interakcję z użytkownikiem w oknie konsoli. Być może będziesz przyjmować dane wejściowe w celu przesłania ich do bazy danych bądź odczytywać liczby wykorzystane w obliczeniach. Bez względu na cel potrzebna nam będzie pętla odczytująca dane wejściowe wpisywane przez użytkownika na klawiaturze i wyświetlająca dla nich wynik. Innymi słowy, musimy utworzyć klasyczną pętlę odczytaj-oblicz-wyswietl.

W Pythonie typowy kod takiej pętli interaktywnej może wyglądać jak poniższy przykład.

```
while True:  
    reply = input('Wpisz dowolny tekst: ')  
    if reply == 'stop': break  
    print(reply.upper())
```

Kod ten wykorzystuje kilka nowych koncepcji oraz kilka rozwiązań, które omawialiśmy już wcześniej:

- W kodzie użyto pętli `while` — najbardziej ogólnej instrukcji pętli Pythona. Instrukcję `while` omówimy bardziej szczegółowo później. Mówiąc w skrócie, zaczyna się ona od słowa `while`, a po nim następuje wyrażenie, którego wynik interpretowany jest jako prawda lub fałsz. Później znajduje się zagnieżdzony blok kodu powtarzany, dopóki test znajdujący się na górze jest prawdą (słowo `True` z przykładu jest zawsze prawdą).
- Wbudowana funkcja `input`, z którą spotkaliśmy się już wcześniej, wykorzystana została tutaj do wygenerowania danych wejściowych z konsoli. Wyświetla ona w charakterze zachęty łańcuch znaków będący opcjonalnym argumentem i zwraca odpowiedź wpisaną przez użytkownika w postaci łańcucha znaków. W wersji 2.x zamiast funkcji `input` powinieneś użyć funkcji `raw_input` (więcej na ten temat już za chwilę).
- W kodzie pojawia się również jednowierszowa instrukcja `if` wykorzystująca regułę specjalną dotyczącą zagnieżdzonych bloków. Ciało instrukcji `if` pojawia się po dwukropku w jej nagłówku, zamiast znajdować się w kolejnym, wciętym wierszu. Oba alternatywne sposoby zadziałają, jednak dzięki metodzie zastosowanej powyżej zaoszczędziliśmy jeden wiersz.
- Instrukcja `break` Pythona wykorzystywana jest do natychmiastowego wyjścia z pętli. Powoduje ona całkowite wyskoczenie z instrukcji pętli i program kontynuowany jest po pętli. Bez takiej instrukcji wyjścia pętla byłaby nieskończona, ponieważ wynik jej testu będzie zawsze prawdziwy.

W rezultacie takie połączenie instrukcji oznacza: „Wczytuj wiersze wpisane przez użytkownika i wyświetlaj je zapisane wielkimi literami, dopóki użytkownik nie wpisze słowa `stop`”. Istnieją

inne sposoby zakodowania takiej pętli, jednak metoda zastosowana powyżej jest w Pythonie często spotykana.

Warto zauważyć, że wszystkie trzy wiersze zagnieżdżone pod wierszem nagłówka instrukcji `while` są wcinane na tę samą odległość. Ponieważ są one wyrównane w pionie jak jedna kolumna, są blokiem kodu powiązanego z testem `while` i powtarzanego. Blok ciała pętli zostaje zakończony albo przez koniec pliku źródłowego, albo przez umieszczenie mniej wciętej instrukcji.

Przykładowe wyniki działania tego kodu po uruchomieniu w sesji interaktywnej lub jako skryptu, zostały pokazane poniżej; kod źródłowy tego przykładu znajduje się w pliku *interact.py* w pakiecie przykładów książki:

Wpisz dowolny tekst:**mielonka**

MIELONKA

Wpisz dowolny tekst:**42**

42

Wpisz dowolny tekst:**stop**



Uwagi na temat wersji: Omawiany przykład jest przygotowany dla Pythona w wersji 3.x. Jeżeli pracujesz w Pythonie 2.x, kod działa tak samo, z tym że zamiast funkcji `input` musisz używać funkcji `raw_input` we wszystkich przykładach tego rozdziału, a we wszystkich wywołaniach instrukcji `print` możesz pominąć nawiasy zewnętrzne (choć ich pozostawienie w niczym nie będzie przeszkadzać). W rzeczywistości, jeżeli przeanalizujesz kod programu *interact.py* w pakiecie przykładów, przekonasz się, że robi to automatycznie — dla zapewnienia zgodności z wersją 2.x podmienia nazwę funkcji, kiedy główna wersja uruchomionego Pythona to 2 (wywołanie `input` kończy się z uruchomieniem funkcji `raw_input`):

```
import sys  
  
if sys.version[0] == '2': input = raw_input # 2.X compatible
```

W wersji 3.x zmieniono nazwę funkcji `raw_input` na `input`, a `print` jest wbudowaną funkcją (więcej na temat funkcji `print` znajdziesz w następnym rozdziale). Python 2.x też zawiera funkcję `input`, która jednak próbuje wykonać dane wejściowe tak, jakby to był kod Pythona, co prawdopodobnie nie będzie działało poprawnie w tym kontekście; w wersji 3.x podobny efekt może dać wywołanie funkcji `eval(input())`.

Wykonywanie obliczeń na danych wpisywanych przez użytkownika

Nasz skrypt działa, jednak teraz założmy, że zamiast zmiany tekstuowego łańcucha znaków na zapisany wielkimi literami wolelibyśmy wykonać jakieś obliczenia na danych liczbowych — na przykład podnosząc je do kwadratu. Aby osiągnąć zamierzony efekt, możemy wypróbować następujący kod:

```
>>> reply = '20'  
>>> reply ** 2  
...pominieto tekst błędu...
```

```
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Takie coś nie zadziała w naszym skrypcie (zgodnie z informacjami z poprzedniej części książki), ponieważ Python nie przekształci typów obiektów w wyrażeniach, jeżeli wszystkie operandy nie są typami liczbowymi — a tak nie jest, ponieważ dane wpisywane przez użytkownika są zawsze w skrypcie zwracane jako łańcuchy znaków. Nie możemy podnieść łańcucha cyfr do potęgi, dopóki ręcznie nie przekształcimy go na liczbę całkowitą.

```
>>> int(reply) ** 2
```

```
400
```

Mając takie informacje, możemy teraz poprawić naszą pętlę w taki sposób, by wykonywała ona niezbędne obliczenia. Aby to sprawdzić, powinieneś umieścić w pliku poniższy kod:

```
while True:
```

```
    reply = input('Wpisz tekst:')  
    if reply == 'stop': break  
    print(int(reply) ** 2)  
print('Koniec')
```

Ten skrypt wykorzystuje jednowierszową instrukcję `if` do wyjścia z pętli w momencie wpisania słowa „stop”, jednak przy okazji konwertuje również dane wejściowe na postać liczbową w celu umożliwienia obliczeń. Wersja ta dodaje także komunikat końcowy umieszczony na dole. Ponieważ instrukcja `print` z ostatniego wiersza nie jest wcięta na tę samą odległość co zagnieżdżony blok kodu, nie jest ona uważana za część ciała pętli i zostanie wykonana tylko raz — po wyjściu z pętli.

```
Wpisz tekst:2
```

```
4
```

```
Wpisz tekst:40
```

```
1600
```

```
Wpisz tekst:stop
```

```
Koniec
```



Jak z tego korzystać: Od tej chwili zakładamy, że ten kod jest zapisywany i uruchamiany z pliku skryptu, za pośrednictwem wiersza polecenia, opcji menu IDLE lub dowolnej innej techniki uruchamiania pliku, którą poznaliśmy w rozdziale 3. W pakiecie przykładów do książki plik ten nosi nazwę `interact.py`. Jeżeli jednak wprowadzasz ten kod interaktywnie, pamiętaj o dołączeniu pustego wiersza (naciśnij dwukrotnie klawisz `Enter`) przed końcową instrukcją wyświetlania, aby zakończyć pętlę. Oznacza to, że nie możesz również wyciąć i wkleić całego kodu do interaktywnego monitu: dodatkowy pusty wiersz wymagany jest w sesji interaktywnej, ale nie w plikach skryptowych. Ostateczne polecenie `print` nie ma jednak sensu w trybie interaktywnym — musisz zakodować go po wejściu w interakcję z pętlą!

Obsługa błędów poprzez sprawdzanie danych wejściowych

Jak na razie wszystko idzie dobrze, ale co się stanie, kiedy dane wejściowe będą niepoprawne?

```
Wpisz tekst:xxx
...pominieto tekst błędu...
ValueError: invalid literal for int() with base 10: 'xxx'
```

Wbudowana funkcja `int` w momencie wystąpienia błędu zwraca tutaj wyjątek. Jeżeli chcemy, by nasz skrypt miał większe możliwości, możemy z wyprzedzeniem sprawdzić zawartość łańcucha znaków za pomocą metody `isdigit` obiektu łańcucha znaków.

```
>>> S = '123'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```

Daje nam to również pretekst do dalszego zagnieżdżenia instrukcji w naszym przykładzie. Poniższa nowa wersja naszego interaktywnego skryptu wykorzystuje pełną instrukcję `if` do obejścia problemu wyjątków pojawiających się w momencie wystąpienia błędu.

```
while True:
    reply = input('Wpisz tekst:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Niepoprawnie!' * 8)
    else:
        print(int(reply) ** 2)
    print('Koniec')
```

Instrukcję `if` przestudujemy szczegółowo w rozdziale 12. Jest ona dość łatwym narzędziem służącym do kodowania logiki w skryptach. W pełnej formie składa się ze słowa `if`, po którym następuje test, powiązany blok kodu, jeden lub większa liczba opcjonalnych testów `elif` (od `else if`) i bloków kodu, a na dole opcjonalna część `else` z powiązanym blokiem kodu, który służy za wynik domyślny. Kiedy pierwszy test zwraca wynik będący prawdą, Python wykonuje blok kodu z nim powiązany — od góry do dołu. Jeżeli wszystkie testy będą zwracały wyniki będące fałszem, wykonywany jest kod z części `else`.

Części `if`, `elif` i `else` w powyższym przykładzie są powiązanymi częściami tej samej instrukcji, ponieważ wszystkie są ze sobą wyrównane w pionie (to znaczy mają ten sam poziom wcięcia). Instrukcja `if` rozciąga się od słowa `if` do początku instrukcji `print` w ostatnim wierszu skryptu. Z kolei cały blok `if` jest częścią pętli `while`, ponieważ w całości wcięty jest pod wierszem nagłówka tej pętli. Zagnieżdżanie instrukcji z czasem stanie się dla każdego naturalne.

Po uruchomieniu kod naszego nowego skryptu wyłapuje błędy przed ich wystąpieniem i wyświetla prosty komunikat o błędzie (który prawdopodobnie będziesz chciał poprawić w kolejnej wersji); wpisanie `stop` nadal kończy działanie pętli, a prawidłowo wpisane liczby są nadal podnoszone do kwadratu:

```
Wpisz tekst:5
```

```
25
```

```
Wpisz tekst:xyz
```

Niepoprawnie! Niepoprawnie! Niepoprawnie! Niepoprawnie! Niepoprawnie!

Wpisz tekst:**10**

100

Wpisz tekst:**stop**

Koniec

Obsługa błędów za pomocą instrukcji try

Powyższe rozwiązywanie działa, jednak, jak zobaczymy w dalszej części książki, najbardziej uniwersalnym sposobem obsługi wyjątków w Pythonie jest przechwytywanie ich i poradzenie sobie z błędem za pomocą instrukcji `try`. Instrukcję tę omówimy bardziej dogłębnie w części VII książki, jednak już teraz możemy pokazać, że użycie tutaj `try` może sprawić, iż kod niektórym osobom wyda się prostszy od poprzedniej wersji.

```
while True:
    reply = input('Wpisz tekst:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Niepoprawnie!' * 5)
    else:
        print(int(reply) ** 2)
    print('Koniec')
```

Ta wersja działa dokładnie tak samo jak poprzednia, jednak zastąpiliśmy dosłowne sprawdzanie błędu kodem zakładającym, że konwersja będzie działała, i opakowaliśmy go kodem z obsługą wyjątku, który zatroszczy się o przypadki, kiedy konwersja nie zadziała.

Instrukcja `try` składa się ze słowa kluczowego `try`, następującego po nim głównego bloku kodu (z działaniem, jakie próbujemy uzyskać), później z części `except` podającej kod obsługujący błędy i części `else`, która jest wykonywana, kiedy żaden wyjątek nie zostanie zgłoszony w części `try`. Python najpierw próbuje wykonać część `try`, a później albo część `except` (jeżeli wystąpi wyjątek), albo `else` (jeżeli wyjątek się nie pojawi).

Jeżeli chodzi o zagnieżdżanie instrukcji, to ponieważ poziom wcięcia `try`, `except` i `else` jest taki sam, wszystkie one uznawane są za część tej samej instrukcji `try`. Warto zauważyć, że część `else` powiązana jest tutaj z `try`, a nie z `if`. Jak widzieliśmy wcześniej, `else` może się w Pythonie pojawiać w instrukcjach `if`, ale także w instrukcjach `try` i pętlach — to wcięcia informują nas, której instrukcji są częścią. W tym przypadku instrukcja `try` rozciąga się od słowa `try` do kodu wciętego pod słowem `else`, ponieważ `else` wcięte jest na tym samym poziomie co `try`. Instrukcja `if` w tym kodzie składa się z jednego wiersza i kończy się po `break`.

Obsługa liczb zmiennoprzecinkowych

Do instrukcji `try` powróćmy w dalszej części książki. Na razie warto być świadomym tego, że ponieważ `try` można wykorzystać do przechwycenia dowolnego błędu, instrukcja ta redukuje ilość kodu sprawdzającego błędy, jaki musimy napisać. Jest także bardzo uniwersalnym

sposobem radzenia sobie z niezwykłymi przypadkami. Na przykład, jeżeli jesteśmy pewni, że funkcja print będzie zawsze działała poprawnie, ten przykład może być jeszcze bardziej zwięzły:

```
while True:  
    reply = input('Wpisz tekst: ')  
    if reply == 'stop': break  
    try:  
        print(int(reply) ** 2)  
    except:  
        print('Niepoprawnie!' * 8)  
    print('Koniec')
```

Jeżeli chcielibyśmy obsługiwać wprowadzanie liczb zmiennoprzecinkowych, zamiast ograniczać się do liczb całkowitych, użycie try byłoby o wiele łatwiejsze od ręcznego testowania błędów — wywołalibyśmy po prostu funkcję float i przechwytywaliśmy wyjątki, zamiast próbować analizować całą potencjalną składnię liczb zmiennoprzecinkowych:

```
while True:  
    reply = input('Wpisz tekst: ')  
    if reply == 'stop': break  
    try:  
        print(float(reply) ** 2)  
    except:  
        print('Niepoprawnie!' * 8)  
    print('Koniec')
```

Obecnie funkcja isfloat dla ciągów nie istnieje, więc podejście oparte na wyjątkach oszczędza nam konieczności analizowania wszystkich możliwych składni zmiennoprzecinkowych w jawnej kontroli błędów. Podczas kodowania w ten sposób możemy wprowadzić szerszy zakres liczb, ale sprawdzanie poprawności danych i zakończenie działania nadal działają jak wcześniej:

```
Wpisz tekst:50  
2500.0  
Wpisz tekst:40.5  
1640.25  
Wpisz tekst 1.23E-100  
1.5129e-200  
Wpisz tekst:spam  
Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!  
!  
Wpisz tekst:stop  
Bye
```



Wywołanie funkcji eval Pythona, którego używaliśmy w rozdziale 5. i rozdziale 9. do konwersji danych w ciągach znaków i plikach, również działałoby w miejscu funkcji float i umożliwiałoby wprowadzanie dowolnych wyrażeń (w takiej sytuacji wyrażenie `2 ** 100` byłoby, co ciekawe, całkowicie poprawnym ciągiem danych wejściowych). Jest to potężna koncepcja, która jest otwarta na te same problemy bezpieczeństwa, o których mowa w poprzednich rozdziałach. Jeżeli źródło, z którego pochodzi kod, nie jest zaufane, powinieneś użyć bardziej restrykcyjnych narzędzi do konwersji, takich jak `int` i `float`.

Funkcja exec Pythona, użyta w rozdziale 3. do uruchomienia kodu odczytanego z pliku, jest podobna do funkcji eval (ale zakłada, że ciąg jest instrukcją zamiast wyrażeniem i nie zwraca wyniku), a wywołanie funkcji compile powoduje wstępne skompilowanie często używanych ciągów kodu do postaci obiektów bytecode, co ma wpływ na zwiększenie szybkości działania. Uruchom polecenie `help` dla dowolnego z nich, aby uzyskać więcej informacji; jak wspomniano, exec jest instrukcją w wersji 2.x, ale funkcją w wersji 3.x. Funkcji exec użyjemy również w rozdziale 25. do importowania modułów według nazw — będzie to dobry przykład użycia tej funkcji w bardziej dynamicznych zastosowaniach.

Kod zagnieżdżony na trzy poziomy głębokości

Przyjrzyjmy się teraz ostatniej mutacji naszego skryptu. Zagnieżdżanie może być jeszcze głębsze — możemy na przykład rozgałęzić jedną z alternatyw w oparciu o względną wielkość poprawnych danych wejściowych.

```
while True:  
    reply = input('Wpisz tekst:')  
    if reply == 'stop':  
        break  
    elif not reply.isdigit():  
        print('Niepoprawnie!' * 8)  
    else:  
        num = int(reply)  
        if num < 20:  
            print('mało')  
        else:  
            print(num ** 2)  
    print('Koniec')
```

Ta wersja zawiera instrukcję `if` zagnieżdzoną w części `else` innej instrukcji `if`, która z kolei jest zagnieżdżona w pętli `while`. Kiedy kod jest warunkowy lub powtarzany, tak jak ten, po prostu wcinamy go jeszcze dalej w prawo. W rezultacie otrzymujemy coś podobnego do poprzedniej wersji, ale dla liczb mniejszych od 20 wyświetlony zostanie komunikat „mało”.

`Wpisz tekst:19`

`mało`

`Wpisz tekst:20`

400

Wpisz tekst:**mielonka**

Niepoprawnie! Niepoprawnie! Niepoprawnie! Niepoprawnie! Niepoprawnie!

Wpisz tekst:**stop**

Koniec

Podsumowanie rozdziału

Powyższe informacje kończą nasze krótkie omówienie podstaw składni instrukcji Pythona. W niniejszym rozdziale wprowadziliśmy ogólne reguły kodowania instrukcji oraz bloków kodu. Jak widzieliśmy, w Pythonie zazwyczaj umieszcza się jedną instrukcję na wiersz i wcina wszystkie instrukcje zagnieżdzonego bloku kodu na tę samą odległość (indentacja jest częścią składni Pythona). Przyjrzelismy się również kilku odstępstwom od tych reguł, w tym wierszom z kontynuacją oraz jednowierszowym testom i pętlom. Wreszcie zastosowaliśmy te koncepcje w praktyce w interaktywnym skrypcie, który zademonstrował kilka instrukcji i pokazał nam, jak tak naprawdę działa składnia instrukcji.

W kolejnym rozdziale zaczniemy zagłębiać się w instrukcje, omawiając bardziej szczegółowo każdą z podstawowych instrukcji proceduralnych Pythona. Jak już jednak widzieliśmy, wszystkie instrukcje zachowują się zgodnie z ogólnymi regułami zaprezentowanymi tutaj.

Sprawdź swoją wiedzę — quiz

1. Jakie trzy elementy wymagane są w językach podobnych do C, ale pomijane w Pythonie?
2. W jaki sposób w Pythonie normalnie kończy się instrukcje?
3. W jaki sposób instrukcje z zagnieżdzonego bloku kodu są zazwyczaj ze sobą wiązane w Pythonie?
4. W jaki sposób możemy rozciągnąć pojedynczą instrukcję na kilka wierszy?
5. W jaki sposób można utworzyć instrukcję złożoną zajmującą jeden wiersz?
6. Czy istnieje jakiś powód uzasadniający umieszczenie średnika na końcu instrukcji w Pythonie?
7. Do czego służy instrukcja `try`?
8. Co jest najczęściej popełnianym przez osoby początkujące błędem w kodowaniu w Pythonie?

Sprawdź swoją wiedzę — odpowiedzi

1. Języki podobne do C wymagają stosowania nawiasów wokół testów w niektórych instrukcjach, średników na końcu instrukcji i nawiasów klamrowych wokół zagnieżdżonych bloków kodu.
2. Koniec wiersza kończy instrukcję umieszczoną w tym wierszu. Alternatywnie, jeżeli w tym samym wierszu znajduje się większa liczba instrukcji, można je zakończyć średnikami. W podobny sposób, kiedy instrukcja rozciąga się na wiele wierszy, trzeba ją zakończyć za pomocą zamknięcia pary nawiasów.
3. Instrukcje w bloku zagnieżdżonym są wcinane o taką samą liczbę tabulatorów lub spacji.
4. Instrukcja może rozciągać się na kilka wierszy dzięki umieszczeniu jej części w nawiasach zwykłych, kwadratowych lub klamrowych. Instrukcja taka kończy się, kiedy Python napotka wiersz zawierający zamykającą część pary nawiasów.
5. Ciało instrukcji złożonej można przesunąć do wiersza nagłówka po dwukropku, jednak tylko wtedy, gdy nie zawiera ono żadnych instrukcji złożonych.
6. Możemy to zrobić tylko wtedy, gdy chcemy zmieścić w jednym wierszu kodu więcej niż jedną instrukcję. Ale nawet wówczas działa to tylko w sytuacji, w której żadna z instrukcji nie jest złożona, i jest odradzane, ponieważ prowadzi do kodu trudniejszego w odczycie.
7. Instrukcja `try` wykorzystywana jest do przechwytywania wyjątków (błędów) i radzenia sobie z nimi w skrypcie Pythona. Zazwyczaj jest alternatywą dla ręcznego sprawdzania błędów w kodzie.
8. Błądem najczęściej popełnianym przez osoby początkujące jest zapominanie o dodaniu dwukropka na końcu wiersza nagłówka instrukcji złożonej. Jeśli ktoś tego błędu nigdy jeszcze nie popełnił, na pewno zrobi to wkrótce!

Rozdział 11. Przypisania, wyrażenia i wyświetlanie

Po krótkim wprowadzeniu do składni instrukcji Pythona niniejszy rozdział rozpoczyna nasze pogłębione omówienie poszczególnych instrukcji tego języka. Zaczniemy od podstaw — przypisania, instrukcji wyrażeń i wyświetlania za pomocą `print`. Ze wszystkimi już się spotkaliśmy, jednak teraz uzupełnimy szczegóły pominięte wcześniej. Choć wszystkie te operacje są stosunkowo proste, jak się niebawem okaże, istnieją opcjonalne odmiany każdego z tych typów instrukcji, które przydadzą się nam przy pisaniu w Pythonie prawdziwych programów.

Instrukcje przypisania

Instrukcji przypisania używamy już od jakiegoś czasu w celu przypisywania obiektów do nazw (zmiennych). W najbardziej podstawowej postaci piszemy *cel* przypisania po lewej stronie znaku równości (=) i *obiekt* przypisywany po prawej stronie. Cel znajdujący się z lewej strony może być nazwą lub komponentem obiektu, natomiast obiekt z prawej strony może być dowolnym wyrażeniem obliczającym obiekt. Najczęściej przypisania są proste, jednak należy pamiętać o kilku kwestiach.

- **Przypisania tworzą referencje do obiektów.** Jak wspomniano w rozdziale 6., w Pythonie przypisania przechowują referencje do obiektów w zmiennych lub komponentach struktur danych. Zawsze tworzą referencje do obiektów, a nie tworzą kopii obiektów. Z tego powodu zmienne Pythona bardziej przypominają wskaźniki niż obszary przechowywania danych.
- **Zmienne tworzone są przy pierwszym przypisaniu.** Python tworzy nazwy zmiennych za pierwszym razem, gdy przypisujemy do nich jakąś wartość (na przykład referencję do obiektu), dlatego nie jest konieczne deklarowanie ich z wyprzedzeniem. Niektóre (ale nie wszystkie) miejsca w strukturach danych tworzone są w momencie przypisania (na przykład wpisy słownika, niektóre atrybuty obiektu). Po przypisaniu zmenna zastępuowana jest wartością, do której się odnosi, za każdym razem, gdy pojawia się ona w wyrażeniu.
- **Przed odwołaniem się do zmiennej trzeba ją najpierw przypisać.** Błądem jest wykorzystywanie zmiennej, do której nie przypisaliśmy jeszcze wartości. Kiedy tak zrobimy, zamiast jakiejś niejednoznacznej wartości domyślnej Python zgłasza wyjątek. W Pythonie okazuje się to kluczowe, ponieważ nazwy nie są zadeklarowane z góry — gdyby program zwracał wartości domyślne dla nieprzypisanych używanych w nim nazw, zamiast traktować je jak błędy, znacznie trudniej byłoby dostrzec błędy w nazwach zmiennych.
- **Niektóre operacje wykonują przypisania niejawne.** W tej części zajmowaliśmy się przede wszystkim instrukcją `=`, jednak przypisania zdarzają się w wielu kontekstach. Jak zobaczymy później, importowanie modułów, definicje funkcji i klas, zmienne pętli `for` i argumenty funkcji są przypisaniami niejawnymi. Ponieważ przypisanie odbywa się w ten sam sposób w każdym miejscu, w chwili uruchomienia takie konteksty po prostu wiążą nazwy (na przykład poprzez przypisanie) z referencjami do obiektów.

Formy instrukcji przypisania

Choć przypisanie jest w Pythonie ogólną i wszechobecną koncepcją, w tym rozdziale interesują nas przede wszystkim *instrukcje przypisania*. W tabeli 11.1 przedstawiono różne formy instrukcji przypisania w Pythonie oraz ich wzorce składniowe.

Tabela 11.1. Formy instrukcji przypisania

Operacja	Interpretacja
<code>spam = 'Mielonka'</code>	Forma podstawowa
<code>spam, ham = 'mniam', 'MNIAM'</code>	Przypisanie krotki (pozycyjne)
<code>[spam, ham] = ['mniam', 'MNIAM']</code>	Przypisanie listy (pozycyjne)
<code>a, b, c, d = 'mielonka'</code>	Przypisanie sekwencji, uogólnione
<code>a, *b = 'mielonka'</code>	Rozszerzone rozpakowanie sekwencji (Python 3.x)
<code>spam = ham = 'lunch'</code>	Przypisanie do wielu celów
<code>spam += 42</code>	Przypisanie rozszerzone (odpowiednik <code>spam = spam + 42</code>)

Pierwsza forma z tabeli 11.1 jest bez wątpienia najczęściej spotykana i polega na połączeniu nazwy (lub komponentu struktury danych) z pojedynczym obiektem. Pozostałe wpisy z tabeli reprezentują formy specjalne i opcjonalne, które w praktyce często przydają się programistom.

Przypisania rozpakowujące krotki i listy

Druga i trzecia forma z tabeli są ze sobą powiązane. Kiedy kodujemy krotki czy listy po lewej stronie znaku `=`, Python łączy obiekty z prawej strony z celami z lewej strony w pary zgodnie z ich pozycją i przypisuje je od lewej do prawej strony. W drugim wierszu tabeli 11.1 nazwa `spam` przypisywana jest do łańcucha znaków '`mniam`', a nazwa `ham` łączona jest z łańcuchem '`MNIAM`'. Wewnętrznie Python najpierw z elementów z prawej strony robi krotkę, dlatego często nazywa się tę operację przypisaniem rozpakowującym krotkę.

Przypisania sekwencji

W nowszych wersjach Pythona przypisania krotki i listy zostały uogólnione jako instancje czegoś, co możemy teraz nazwać wywołaniem *przypisania sekwencji*. Dowolną sekwencję można przypisać do dowolnej sekwencji wartości, a Python przypisuje elementy po jednym na raz zgodnie z ich pozycją. Tak naprawdę możemy mieszać i dopasowywać typy wykorzystywanych sekwencji. W czwartym wierszu tabeli 11.1 w parę połączono krotkę nazw i łańcuch znaków — a zostaje przypisane do '`s`', `b` do '`p`' i tak dalej.

Rozszerzone rozpakowanie sekwencji

W Pythonie 3.x (wyłącznie) wprowadzono nową formę przypisania sekwencji, która daje więcej możliwości wyboru fragmentu sekwencji do przypisania. W przykładzie z piątego wiersza tabeli 11.1, zmiennej `a` zostanie przypisana litera `m`, natomiast zmiennej `b` — ciąg znaków `['i', 'e', 'l', 'o', 'n', 'k', 'a']`. Rozszerzona składnia rozpakowania sekwencji upraszcza tego typu operacje, pozwalając uniknąć stosowania wycinków w przypisaniach.

Przypisania do wielu celów

Piąty wiersz tabeli 11.1 prezentuje formę przypisania z wieloma celami. W tej formie Python przypisuje referencję do jednego obiektu (znajdującego się najdalej na prawo) do wszystkich celów znajdujących się po lewej stronie. W tabeli do nazw `spam` i `ham` przypisane są referencje do tego samego obiektu łańcucha znaków, 'lunch'. Rezultat jest taki sam, jakbyśmy w kodzie napisali `ham = 'lunch'`, a potem `spam = ham`, ponieważ `ham` ma wartość oryginalnego obiektu łańcucha znaków (a nie osobnej kopii tego obiektu).

Przypisania rozszerzone

Ostatni wiersz tabeli 11.1 jest przykładem *przypisania rozszerzonego* — skrótu łączącego wyrażenie i przypisanie w zwięzły sposób. Przypisanie `spam += 42` ma na przykład ten sam efekt co `spam = spam + 42`, jednak forma rozszerzona wymaga mniej pisania i jest generalnie szybsza do wykonania. Dodatkowo w przypadku, gdy obiekt jest mutowalny i obsługuje daną operację, przypisania rozszerzone mogą działać szybciej dzięki zastosowaniu modyfikacji obiektu w miejscu zamiast jego kopiowania. Jak się niebabem przekonasz, w Pythonie większość operatorów wyrażeń binarnych obsługuje przynajmniej jedno przypisanie rozszerzone.

Przypisanie sekwencji

W książce korzystaliśmy już z podstawowej formy przypisania, więc możemy przyjąć, że wiesz już, o co w tym chodzi. Poniżej znajduje się kilka prostych przykładów przypisania rozpakowującego sekwencje.

```
% python
>>> nudge = 1                                # Podstawowe przypisanie
>>> wink = 2
>>> A, B = nudge, wink                      # Przypisanie krotki
>>> A, B                                      # Jak A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink]                   # Przypisanie listy
>>> C, D
(1, 2)
```

Zwróć uwagę, że w trzecim wierszu tego kodu tak naprawdę tworzymy dwie krotki — pomijamy tylko ich nawiasy. Python łączy w pary wartości z krotki z prawej strony operatora przypisania z wartościami z krotki z lewej strony i przypisuje po jednej wartości na raz.

Przypisanie krotek prowadzi do często stosowanej sztuczki wprowadzonej w rozwiązańach do ćwiczeń z drugiej części książki. Ponieważ Python tworzy tymczasową krotkę, która zapisuje oryginalne wartości zmiennych z prawej strony w czasie wykonywania instrukcji, przypisania tego typu służą również do zamiany wartości dwóch zmiennych bez tworzenia osobnej zmiennej tymczasowej — krotka z prawej strony automatycznie pamięta poprzednie wartości zmiennych.

```
>>> nudge = 1
>>> wink = 2
>>> nudge, wink = wink, nudge                 # Krotki: zamiana wartości
>>> nudge, wink                               # Jak T = nudge; nudge = wink;
wink = T
(2, 1)
```

Tak naprawdę oryginalne formy przypisania krotek i list w Pythonie zostały z czasem uogólnione w taki sposób, by przyjmowały dowolny typ sekwencji po prawej stronie, pod warunkiem, że długość sekwencji będzie taka sama. Krotkę wartości można na przykład przypisać do listy zmiennych, a łańcuch znaków do krotki zmiennych. W każdym przypadku Python przypisuje elementy z sekwencji po prawej stronie do zmiennych z sekwencji po lewej stronie zgodnie z ich pozycją, od lewej do prawej.

```
>>> [a, b, c] = (1, 2, 3)                                # Przypisanie krotki wartości do
listy nazw

>>> a, c
(1, 3)

>>> (a, b, c) = "ABC"                                    # Przypisanie łańcucha znaków do
krotki

>>> a, c
('A', 'C')
```

Z technicznego punktu widzenia przypisanie sekwencji tak naprawdę akceptuje po prawej stronie wyrażenia dowolny obiekt *iterowalny*, nie tylko sekwencję. Jest to bardziej ogólna kategoria, która obejmuje kolekcje zarówno fizyczne (np. listy), jak i wirtualne (np. wiersze danych w pliku), które zostały krótko zdefiniowane w rozdziale 4. i pojawiają się od tego czasu. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 14. i rozdziale 20., gdzie będziemy omawiać iteracje.

Zaawansowane wzorce przypisywania sekwencji

Jedna uwaga: choć możemy mieszać i dopasowywać typy sekwencji znajdujące się wokół symbolu `=`, nadal musimy mieć *tą samą liczbę* elementów po prawej stronie co zmiennych po lewej stronie. W przeciwnym razie otrzymamy błąd. Python 3.x pozwala zachować wyższy poziom uogólnienia dzięki rozszerzonej składni rozpakowania, opisanej w następnym punkcie rozdziału. Z reguły jednak w wersji 3.x (a zawsze w 2.x) liczba zmiennych po obydwu stronach wyrażenia powinna się zgadzać.

```
>>> string = 'JAJO'

>>> a, b, c, d = string                                # Ta sama liczba elementów po
obu stronach

>>> a, d
('J', 'O')

>>> a, b, c = string                                    # Błąd: różna liczba elementów
...pominieto tekst błędu...

ValueError: too many values to unpack (expected 3)
```

W celu uzyskania bardziej elastycznego rozwiązania zarówno w wersji 2.x, jak i 3.x możemy skorzystać z wycinków. Istnieje kilka sposobów zastosowania wycinków w taki sposób, aby ostatni przykład działał poprawnie:

```
>>> a, b, c = string[0], string[1], string[2:] # Indeksowanie i wycinek

>>> a, b, c
('J', 'A', 'JO')

>>> a, b, c = list(string[:2]) + [string[2:]] # Wycinek i konkatenacja
```

```

>>> a, b, c
('J', 'A', 'JO')
>>> a, b = string[:2]                                # To samo w prostszej formie
>>> c = string[2:]
>>> a, b, c
('J', 'A', 'JO')
>>> (a, b), c = string[:2], string[2:]            # Zagnieżdżone sekwencje
>>> a, b, c
('J', 'A', 'JO')

```

W ostatnim przykładzie tego kodu widać, że możemy nawet przypisać sekwencje *zagnieżdżone*, ponieważ Python, zgodnie z oczekiwaniami, rozpakowuje ich części, uwzględniając ich kształt. W tym przypadku przypisujemy krotkę dwuelementową dokładnie w taki sposób, w jaki byśmy ją zakodowali; pierwszym elementem jest zagnieżdżona sekwencja (łańcuch znaków).

```

>>> ((a, b), c) = ('JA', 'JO')                      # Połączone w pary zgodnie z
ksztaltem i pozycją
>>> a, b, c
('J', 'A', 'JO')

```

Python łączy pierwszy łańcuch znaków z prawej strony ('JA') z pierwszą krotką z lewej strony ((a, b)) i przypisuje po jednym znaku na raz przed przypisaniem całego drugiego łańcucha znaków ('JO') do zmiennej c za jednym razem. W takiej sytuacji kształt zagnieżdżenia sekwencji z lewej strony musi odpowiadać obiektowi z prawej strony. Takie przypisania zagnieżdżonych sekwencji są nieco bardziej zaawansowane i rzadziej się je spotyka, jednak mogą być wygodne dla celów wyboru części struktur danych o znanym kształcie.

Technika ta działa na przykład również w pętlach `for`, ponieważ elementy pętli są przypisane do celu podanego w nagłówku pętli, o czym będziemy mówić w rozdziale 13. książki:

```

for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...          # Proste przypisanie do
krotki

for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ...    # Zagnieżdżone
przypisanie do krotki

```

W uwadze w rozdziale 18. napisałem, że przypisania rozpakowujące zagnieżdżone krotki (a w praktyce sekwencje) działają również w przypadku list argumentów funkcji w wersji 2.x (ale już nie w 3.x), ponieważ argumenty funkcji również są przekazywane przez przypisania.

```

def f(((a, b), c)):                                # Również dla argumentów w Pythonie 2.x, ale
nie 3.x

f(((1, 2), 3))

```

Przypisania rozpakowujące sekwencje spowodowały również pojawienie się kolejnej sztuczki programistycznej w Pythonie — przypisania serii liczb całkowitych do zbioru zmiennych.

```

>>> red, green, blue = range(3)
>>> red, blue
(0, 2)

```

Powyższy kod inicjalizuje trzy zmienne i nadaje im wartości liczb całkowitych 0, 1 oraz 2 (to odpowiednik *wyliczeniowych* typów danych obecnych w innych językach programowania). Aby zrozumieć ten przykład, trzeba wiedzieć, że wbudowana funkcja `range` generuje listę kolejnych liczb całkowitych (jeżeli w wersji 3.x chcesz wyświetlić wszystkie utworzone wartości naraz, musisz przekazać funkcję `range` jako argument wywołania funkcji `list`):

```
>>> list(range(3))          # list() jest wymagane tylko w Pythonie 3.x
[0, 1, 2]
```

Taki sposób wywołania omawialiśmy pokrótko w rozdziale 4.; ze względu jednak na fakt, że funkcja `range` jest często wykorzystywana w pętlach `for`, więcej powiemy o niej jeszcze w rozdziale 13.

Kolejnym miejscem, w którym wykorzystuje się przypisanie krotek, jest dzielenie sekwencji w pętlach na jej przód i resztę, jak w poniższym kodzie.

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, L = L[0], L[1:]      # Wersja dla 3.x przedstawiona w następnym
...                                 # punkcie
...     print front, L
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Przypisanie krotki w pętli mogłoby również zostać zakodowane w poniższych dwóch wierszach, jednak na ogół wygodniej jest połączyć je ze sobą.

```
...     front = L[0]
...     L = L[1:]
```

Warto zwrócić uwagę na to, że ten kod wykorzystuje listę jako swoistą strukturę danych stosu — coś, co często uzyskujemy za pomocą metod `append` i `pop` obiektu listy. Tutaj `front = L.pop(0)` miałoby mniej więcej ten sam efekt co instrukcja z przypisaniem krotki, jednak byłaby to modyfikacja w miejscu. W rozdziale 13. dowiemy się więcej na temat pętli `while` i innych (często lepszych) sposobów przechodzenia sekwencji za pomocą pętli `for`.

Rozszerzona składnia rozpakowania sekwencji w Pythonie 3.x

W poprzednim punkcie zademonstrowałem, w jaki sposób używać wycinania fragmentów sekwencji, aby uogólnić operacje ich przypisania. W Pythonie 3.x (ale nie w 2.x) operacje przypisania sekwencji zostały uogólnione, co upraszcza to zadanie. W skrócie: w sekwencji celu przypisania można użyć pojedynczej *nazwy z gwiazdką* (`*X`), której zostanie przypisana lista elementów nieprzypisanych innym zmiennym. Taka możliwość przydaje się szczególnie w przypisaniach typu „pierwsze X elementów i reszta”, jak w ostatnim przykładzie poprzedniego punktu.

Rozszerzona składania rozpakowania w działaniu

Rzućmy okiem na przykład. Jak widzieliśmy, przypisania sekwencji wymagają z reguły tej samej liczby zmiennych po lewej stronie przypisania, ile elementów zawiera sekwencja z prawej strony. Jeżeli liczby elementów nie są zgodne, przypisanie zakończy się błędem (zarówno w wersji 2.x, jak i 3.x), co widzieliśmy również w poprzednim punkcie.

```
C:\code> c:\python33\python
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> print(a, b, c, d)
1 2 3 4
>>> a, b = seq
ValueError: too many values to unpack (expected 2)
```

W Pythonie 3.x jedna z nazw po lewej stronie przypisania może być poprzedzona gwiazdką, co daje dodatkowe możliwości. Poniższy listing stanowi kontynuację rozpoczętej wyżej sesji interaktywnej, czyli rozpakowujemy tę samą sekwencję. Zmiennej **a** przypisujemy pierwszy element sekwencji, zmiennej **b** listę pozostałych elementów:

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

W przypadku użycia nazwy z gwiazdką liczba zmiennych po lewej stronie przypisania nie musi zgadzać się z liczbą elementów sekwencji po prawej. W rzeczywistości nazwa z gwiazdką może wystąpić w dowolnym miejscu sekwencji. Na przykład w następnej iteracji zmiennej **b** przypisujemy ostatni element sekwencji, a zmiennej **a** wszystkie jej elementy oprócz ostatniego:

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

Jeżeli nazwa z gwiazdką wystąpi w środku, zostaną jej przypisane wszystkie elementy z pominięciem tych, które zostały przypisane zmiennym bez gwiazdki. W poniższym przykładzie **a** i **c** otrzymują odpowiednio pierwszy i ostatni element sekwencji, zmiennej **b** zostaje przypisana kopia sekwencji z pominięciem pierwszego i ostatniego elementu.

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
```

A więc w przypadku, gdy w przypisaniu sekwencji występuje nazwa z gwiazdką, zostaną jej przypisane wszystkie elementy, które nie zostały przypisane do pozostałych zmiennych.

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

Oczywiście podobnie jak zwykłe przypisanie sekwencji, tak i rozszerzona składnia rozpakowania sekwencji działa z dowolnymi typami sekwencyjnymi, nie tylko z listami (w rzeczywistości, podobnie jak poprzednio, z dowolnymi typami *iterowalnymi*). Poniższy listing prezentuje przykład rozpakowania ciągu znaków i zakresu range (obiekt iterowalny w Pythonie 3.x).

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])
>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm')
>>> a, *b, c = range(4)
>>> a, b, c
(0, [1, 2], 3)
```

Reguła ta przypomina tworzenie wycinków, ale to nie jest dokładnie to samo: przypisanie rozpakowujące sekwencję zawsze zwraca listę, natomiast tworzenie wycinków zwraca obiekt tego samego typu, co obiekt poddany operacji wycinania.

```
>>> S = 'spam'
>>> S[0], S[1:] # Wycinki są zależne od typu, przypisanie *
zawsze zwraca listę
('s', 'pam')
>>> S[0], S[1:3], S[3]
('s', 'pa', 'm')
```

Dzięki temu rozszerzeniu Pythona 3.x ostatni przykład poprzedniego punktu staje się jeszcze prostszy, ponieważ nie musimy ręcznie przypisywać wycinka, aby uzyskać listę nieprzypisanych elementów sekwencji.

```
>>> L = [1, 2, 3, 4]
>>> while L:
```

```

...      front, *L = L          # Pobieramy pierwszy element i listę
pozostałych, bez wycinania
...      print(front, L)

...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []

```

Przypadki brzegowe

Składnia rozszerzonego rozpakowania sekwencji jest elastycznym mechanizmem, ale warto pamiętać o szczególnych przypadkach. Po pierwsze, nazwa z gwiazdką może otrzymać pojedynczy element, ale zawsze będzie to lista:

```

>>> seq = [1, 2, 3, 4]
>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]

```

Po drugie, w sytuacji, gdy nic nie zostanie, nazwie z gwiazdką zostaje przypisana pusta lista, niezależnie od tego, w którym miejscu listy zmiennych występuje. W poniższym przykładzie zmiennym a, b, i c przypisano wszystkie elementy sekwencji, ale Python przypisuje pustą listę zmiennej e i nie wywołuje błędu.

```

>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []

```

Błędy mogą jednak wystąpić, jeżeli w przypisaniu występuje większa liczba nazw z gwiazdką, albo jeżeli wystąpi za duża liczba zmiennych bez gwiazdki (czyli analogicznie do zwykłego przypisania sekwencji) oraz w przypadku, gdy nazwa z gwiazdką zostanie użyta poza wyrażeniem przypisania.

```

>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment
>>> a, b = seq
ValueError: too many values to unpack (expected 2)
>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple
>>> *a, = seq
>>> a

```

```
[1, 2, 3, 4]
```

Wygodny gadżet

Pamiętaj, że rozszerzona składnia rozpakowania sekwencji służy jedynie zwiększeniu wygody programisty. Można sobie bez niej poradzić (a w 2.x trzeba), wykorzystując odczyt po indeksie i tworzenie wycinków, ale składnia rozpakowująca jest łatwiejsza w użyciu. Opisany wzorzec „pierwszy element i reszta” jest dość powszechny w programowaniu i można go zaimplementować na wiele sposobów, ale użycie wycinków wymaga dodatkowych wierszy kodu:

```
>>> seq
[1, 2, 3, 4]
>>> a, *b = seq                         # Pierwszy, pozostałe
>>> a, b
(1, [2, 3, 4])
>>> a, b = seq[0], seq[1:]               # Pierwszy, pozostałe: metoda
                                         tradycyjna
>>> a, b
(1, [2, 3, 4])
```

Również powszechny wzorzec „początek, ostatni element” implementuje się podobnie, ale użycie składni rozpakowującej wymaga zauważalnie mniejszej ilości kodu:

```
>>> *a, b = seq                         # Początek, ostatni
>>> a, b
([1, 2, 3], 4)
>>> a, b = seq[:-1], seq[-1]            # Początek, ostatni: metoda tradycyjna
>>> a, b
([1, 2, 3], 4)
```

Użycie rozszerzonej składni rozpakowania sekwencji jest prostsze i wydaje się bardziej naturalne, co wróży tej nowości w Pythonie zdobycie z czasem dużej popularności.

Zastosowanie w pętli for

Zmienna użyta w pętli `for` może być określona z użyciem dowolnego przypisania, zatem i w tym przypadku działa przypisanie sekwencji. Z iteracjami w pętli `for` spotkaliśmy się już w rozdziale 4., a ich pełnej analizie poświęcimy rozdział 13. W Pythonie 3.x składania rozszerzonego przypisania może wystąpić po słowie kluczowym `for`, gdzie najczęściej stosuje się zwykłe przypisanie do pojedynczej zmiennej:

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    ...
```

W tym kontekście w każdej iteracji Python przypisuje zmiennym `a`, `b` i `c` kolejną krotkę wartości. Na przykład w pierwszym przebiegu pętli zmienne te otrzymują takie wartości, jak w wyniku następującego wywołania:

```
a, *b, c = (1, 2, 3, 4)                  # b ma wartość [2, 3]
```

Nazwy `a`, `b` i `c` mogą być użyte w kodzie pętli, udostępniając rozpakowane elementy krotki. W rzeczywistości takie użycie nie jest specjalnym przypadkiem, ponieważ stanowi jeden z

przypadków wyrażeń przypisania. Jak widzieliśmy wcześniej w tym rozdziale, możemy dokonać podobnego przypisania do krotek, które działa w Pythonie 2.x i 3.x:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: # a, b, c = (1, 2, 3), ...
```

Oczywiście i w tym kontekście w Pythonie 2.x możemy zaemulować rozszerzoną składnię rozpakowania sekwencji dostępną w 3.x, używając wycinków:

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
    a, b, c = all[0], all[1:3], all[3]
```

Na razie nie posiadamy wystarczającej wiedzy na temat tego typu składni w pętlach `for`, zatem odłożymy omawianie tego tematu do rozdziału 13.

Przypisanie z wieloma celami

Przypisanie z wieloma celami po prostu przypisuje wszystkie podane zmienne do obiektu znajdującego się najbardziej po prawej stronie. Poniższy kod przypisuje na przykład trzy zmienne (`a`, `b` oraz `c`) do łańcucha znaków '`mielonka`'.

```
>>> a = b = c = 'mielonka'  
>>> a, b, c  
('mielonka', 'mielonka', 'mielonka')
```

Forma ta jest (łatwiejszym do zapisania) odpowiednikiem poniższych trzech instrukcji przypisania.

```
>>> c = 'mielonka'  
>>> b = c  
>>> a = b
```

Przypisanie z wieloma celami a współdzielone referencje

Pamiętaj, że ciągle mamy tu jeden obiekt — współdzielony przez wszystkie trzy zmienne (wszystkie one wskazują na ten sam obiekt w pamięci). Takie zachowanie jest poprawne w przypadku typów niemutowalnych — na przykład inicjalizacji zbioru liczników do wartości zero (przypomnijmy, że przed użyciem zmienne muszą zostać w Pythonie przypisane, dlatego przed dodaniem czegoś do liczników trzeba je najpierw zainicjalizować do wartości zero).

```
>>> a = b = 0  
>>> b = b + 1  
>>> a, b  
(0, 1)
```

Tutaj zmiana `b` modyfikuje jedynie `b`, ponieważ liczby nie obsługują modyfikacji w miejscu. Dopóki przypisywany obiekt jest niemutowalny, fakt, że większa liczba zmiennych zawiera referencje do niego, nie ma większego znaczenia.

Jak zawsze jednak musimy być ostrożni, kiedy inicjalizujemy zmienne do pustych obiektów zmiennych, takich jak lista czy słownik.

```
>>> a = b = []  
>>> b.append(42)
```

```
>>> a, b  
([42], [42])
```

Tym razem, ponieważ zmienne `a` i `b` zawierają referencje do tego samego obiektu, dodanie do niego jakiegoś elementu w miejscu za pośrednictwem zmiennej `b` będzie miało również wpływ na to, co zobaczymy w `a`. To tak naprawdę kolejny przykład zjawiska współdzielonych referencji, z którym pierwszy raz spotkaliśmy się w rozdziale 6. By uniknąć tego problemu, należy zamiast tego inicjalizować obiekty zmienne w osobnych instrukcjach, wykonując oddzielne wyrażenie z literałem, tak by każdy z nich utworzył osobny pusty obiekt.

```
>>> a = []  
>>> b = []  
                  # a i b nie współdzielą tego  
samego obiektu  
>>> b.append(42)  
>>> a, b  
([], [42])
```

Przypisanie krotek takie jak poniżej daje ten sam efekt — uruchamiając dwa wyrażenia listy, tworzymy dwa różne obiekty:

```
>>> a, b = [], []  
                  # a i b nie współdzielą tego samego obiektu
```

Przypisania rozszerzone

Od Pythona 2.0 udostępniony został zbiór dodatkowych formatów instrukcji przypisania wymieniony w tabeli 11.2. Znane jako *przypisania rozszerzone* i zapożyczone z języka C, formaty te są generalnie skrótami. Są one kombinacją wyrażenia binarnego i przypisania. Poniższe dwa formaty są na przykład mniej więcej równoważne.

```
X = X + Y  
X += Y  
                  # Forma tradycyjna  
                  # Nowsza forma rozszerzona
```

Tabela 11.2. Rozszerzone instrukcje przypisania

X += Y	X &= Y	X -= Y	X = Y
X *= Y	X ^= Y	X /= Y	X >>= Y
X %= Y	X <<= Y	X **= Y	X //= Y

Przypisania rozszerzone działają na dowolnym typie obiektów obsługujących wyrażenia binarne. Poniżej widać dwa sposoby dodania liczby 1 do zmiennej.

```
>>> x = 1  
>>> x = x + 1  
                  # Forma tradycyjna  
>>> x  
2  
>>> x += 1  
                  # Forma rozszerzona  
>>> x  
3
```

Po zastosowaniu do sekwencji takich jak łańcuchy znaków forma rozszerzona wykonuje zamiast tego konkatenację. Z tego powodu drugi wiersz poniższego kodu jest odpowiednikiem wpisania nieco dłuższego polecenia `S = S + "MIELONKA"`.

```
>>> S = "mielonka"  
>>> S += "MIELONKA" # Niejawnia konkatenacja  
>>> S  
'mielonkaMIELONKA'
```

Jak widać w tabeli 11.2, istnieją analogiczne formy rozszerzonego przypisania dla większości operatorów wyrażeń binarnych Pythona (czyli dla operatorów z wartościami po lewej i prawej stronie). Przykładowo `X *= Y` mnoży i przypisuje, a `X >= Y` przesuwa w prawo i przypisuje. Instrukcja `X // Y` (dzielenie bez reszty) została dodana w Pythonie 2.2.

Przypisania rozszerzone mają trzy zalety:^[1]

- Mamy mniej kodu do wpisania. Czy trzeba dodawać coś więcej?
- Lewa strona musi być obliczona tylko raz. W `X += Y` zmienna `X` może być skomplikowanym wyrażeniem obiektu. W formie rozszerzonej wyrażenie to musi być obliczone tylko raz. W dłuższej formie (`X = X + Y`) zmienna `X` pojawia się dwa razy i musi też być dwa razy wykonana. Z tego powodu przypisania rozszerzone są zazwyczaj szybsze.
- Optymalna technika wybierana jest automatycznie. W przypadku obiektów obsługujących modyfikację w miejscu formy rozszerzone automatycznie wykonują operacje modyfikacji w miejscu zamiast wolniejszych kopii.

Ostatni z powyższych punktów zasługuje na słowo wyjaśnienia. W przypadku przypisania rozszerzonego w obiektach zmiennych w celu optymalizacji mogą być zastosowane operacje modyfikujące obiekt w miejscu. Warto sobie przypomnieć, że listy mogą być rozszerzane na kilka sposobów. Aby dodać pojedynczy element na końcu listy, możemy skorzystać z konkatenacji lub metody `append`.

```
>>> L = [1, 2]  
>>> L = L + [3] # Konkatenacja: wolniej  
>>> L  
[1, 2, 3]  
>>> L.append(4) # Szybciej, ale w miejscu  
>>> L  
[1, 2, 3, 4]
```

Aby dodać zbiór elementów na końcu, możemy albo znowu skorzystać z konkatenacji, albo wywołać metodę `extend` listy^[2].

```
>>> L = L + [5, 6] # Konkatenacja: wolniej  
>>> L  
[1, 2, 3, 4, 5, 6]  
>>> L.extend([7, 8]) # Szybciej, ale w miejscu  
>>> L  
[1, 2, 3, 4, 5, 6, 7, 8]
```

W obu przypadkach konkatenacja jest mniej podatna na efekty uboczne współdzielonych referencji do obiektu, jednak zazwyczaj będzie działała wolniej od swojego odpowiednika wykonującego modyfikację w miejscu. Operacje konkatenacji muszą utworzyć nowy obiekt, skopiować listę z lewej strony, a następnie skopiować listę z prawej strony. Modyfikacje w miejscu za pomocą wywołania metod po prostu dodają elementy na końcu bloku pamięci (wewnętrznie jest to zwykle nieco bardziej skomplikowane, ale dla naszych potrzeb taki opis w zupełności wystarczy).

Kiedy do rozszerzenia listy wykorzystamy rozszerzoną formę przypisania, możemy zapomnieć o tych szczegółach — na przykład Python automatycznie wywołuje szybszą metodę `extend`, zamiast korzystać z wolniejszej operacji konkatenacji, którą sugerowałoby użycie operatora `+`.

```
>>> L += [9, 10] # Odwzorowane na L.extend([9, 10])  
>>> L  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Zauważ jednak, że z powodu tej równoważności `+=` dla listy nie jest dokładnie taki sam jak `+ i =` we wszystkich przypadkach — dla list przypisanie rozszerzone `+=` pozwala na użycie dowolnych sekwencji (podobnie jak `extend`), ale z konkatenacją normalnie tak nie jest:

```
>>> L = []
>>> L += 'spam'          # += oraz extend zezwalają na użycie dowolnej
sekwencji, ale z operatorem + tak nie jest!
>>> L
['s', 'p', 'a', 'm']
>>> L = L + 'spam'
TypeError: can only concatenate list (not "str") to list
```

Przypisy rozszerzone a współdzielone referencje

Zazwyczaj takiego zachowania właśnie sobie życzymy, jednak warto zauważyć, że sugeruje ono, iż `+=` jest modyfikacją listy w miejscu. Z tego powodu nie do końca przypomina konkatenację z operatorem `+`, która zawsze tworzy nowy obiekt. Jeżeli chodzi o referencje współdzielone, różnica ta może mieć znaczenie, gdy różne zmienne wskazują na modyfikowany obiekt.

```
([1, 2, 3, 4], [1, 2, 3, 4])
```

Ma to znaczenie jedynie w przypadku obiektów mutowalnych, takich jak listy i słowniki, i jest stosunkowo egzotycznym przypadkiem (dopóki, oczywiście, nie zacznie nam przysparzać problemów w kodzie). Jak zawsze, kiedy chcesz przerwać strukturę referencji współdzielonych, powinieneś utworzyć kopię obiektów mutowalnych.

Reguły dotyczące nazw zmiennych

Skoro omówiliśmy już instrukcje przypisania, czas zająć się nazwami zmiennych z bardziej formalnego punktu widzenia. W Pythonie zmienne pojawiają się, kiedy przypisuje się do nich wartości, jednak istnieje kilka reguł dotyczących wyboru nazw dla komponentów naszego programu.

Składnia: (znak podkreślenia lub litera) + (dowolna liczba liter, cyfr i znaków podkreślenia)

Nazwy zmiennych muszą rozpoczynać się od liter lub znaków podkreślenia, po których może nastęować dowolna liczba liter, cyfr lub znaków podkreślenia. W Pythonie poprawnymi nazwami są `_mielonka`, `mielonka` i `Mielonka_1`, natomiast `1_Mielonka`, `mielonka$` i `@#!` są niepoprawne.

Wielkość liter ma znaczenie — MIELONKA to nie to samo co mielonka

W programach Python zawsze zwraca uwagę na wielkość liter — zarówno w tworzonych nazwach, jak i w słowach zarezerwowanych. Nazwy `X` i `x` odnoszą się do dwóch różnych zmiennych. W przypadku przenośności wielkość liter ma znaczenie również w nazwach importowanych plików modułów nawet na platformach, w których systemach plików wielkość liter nie ma znaczenia. W ten sposób importowanie nadal działa, gdy programy są uruchamiane na różnych platformach.

Słowa zarezerwowane nie mogą być stosowane

Definiowane nazwy nie mogą być takie same jak słowa zarezerwowane mające w Pythonie specjalne znaczenie. Jeżeli na przykład spróbujesz użyć nazwy zmiennej takiej jak `class`, Python zwróci błąd składni; `Klass` i `Class` będą poprawne. W tabeli 11.3 wymieniono aktualne słowa zarezerwowane Pythona.

Tabela 11.3. Słowa zarezerwowane Pythona 3.x

<code>False</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>None</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>True</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
<code>and</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
<code>as</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>
<code>assert</code>	<code>else</code>	<code>import</code>	<code>pass</code>	
<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>	

Tabela 11.3 zawiera słowa kluczowe Pythona 3.x. W Pythonie 2.x zestaw zarezerwowanych słów kluczowych odróżni się różni:

- `print` jest słowem zarezerwowanym, ponieważ jest instrukcją, nie funkcją wbudowaną (więcej na ten temat w dalszej części tego rozdziału).

- `exec` jest słowem zarezerwowanym, ponieważ jest instrukcją, a nie funkcją wbudowaną.
- `nonlocal` nie jest zarezerwowanym słowem, ponieważ ta instrukcja nie jest dostępna.

W starszych wersjach Pythona jest podobnie, z kilkoma różnicami:

- `with` oraz `as` nie były zarezerwowane do wersji 2.6, kiedy wprowadzono mechanizm menedżera kontekstu.
- `yield` nie było nazwą zarezerwowaną do wersji 2.3, kiedy wprowadzono funkcje generatorów.
- `yield` w wersji 2.5 z instrukcji przekształcono w wyrażenie, ale wciąż jest słowem zarezerwowanym, a nie wbudowaną funkcją.

Jak łatwo zauważyc, słowa zarezerwowane Pythona pisane są małymi literami i naprawdę są one zarezerwowane — w przeciwieństwie do nazw wbudowanych, z którymi spotkamy się w kolejnej części książki, nie możemy redefiniować słów zarezerwowanych za pomocą przypisania (na przykład polecenie `and = 1` spowoduje zwrócenie błędu składni) [3].

Pierwsze trzy słowa kluczowe przedstawione w tabeli 11.3, `True`, `False` i `None`, są szczególnie nie tylko dlatego, że rozpoczynają się wielką literą. Są one bowiem elementami wbudowanego zakresu nazw Pythona, co zostało szczegółowo opisane w rozdziale 17., i z technicznego punktu widzenia są nazwami związanymi z konkretnymi obiektami. W Pythonie 3.x są jednak nazwami naprawdę zarezerwowanymi pod każdym względem i nie można ich użyć w innym kontekście niż przewidziany dla związań z nimi obiektów. Wszystkie pozostałe słowa zarezerwowane są natomiast elementami składni języka Python i mogą wystąpić tylko w kontekstach instrukcji, do których zostały przeznaczone.

Co więcej, ponieważ nazwy modułów w instrukcjach `import` stają się zmiennymi w skrypcie, ograniczenie to rozciąga się również na nazwy plików modułów — możemy utworzyć pliki o nazwach `and.py` i `my-code.py`, jednak nie możemy ich zimportować. Ponieważ ich nazwy bez rozszerzenia `.py` stają się *zmiennymi* w kodzie, muszą być zgodne z regułami przedstawionymi dla zmiennych (słowa zarezerwowane nie mogą zostać użyte, a myślniki nie działają — zamiast nich można jednak użyć znaków `_`). Powróćmy do tej kwestii w piątej części książki.

Protokół przedawnienia w Pythonie

Interesująca jest obserwacja procesu stopniowego wprowadzania zarezerwowanych słów kluczowych do języka. Gdy nowa funkcja może popsuć istniejący kod, w Pythonie z reguły zmianę wprowadza się stopniowo z zastosowaniem mechanizmu „przedawnienia” (ang. *deprecation*), czyli wyświetlania ostrzeżeń w przypadku użycia takich konstrukcji, które mają być wycofane z języka. Mechanizm przedawnienia opiera się na założeniu, że programista powinien mieć czas na zauważenie ostrzeżeń i aktualizację kodu przed dokonaniem migracji do nowego wydania Pythona. W przypadku dużych wydań, jak 3.0, mechanizm ten nie jest stosowany, przez co ryzyko niekompatybilności jest niemałe, ale jest stosowany od wielu lat w przypadku zwykłych aktualizacji wersji.

Na przykład `yield`: w Pythonie 2.2 był opcjonalnym rozszerzeniem, a w 2.3 stał się standardowym słowem kluczowym. To słowo kluczowe jest stosowane w funkcjach generatorów. Jest to jeden z licznych przykładów, gdy Python puści zgodność ze starym kodem. Jednak `yield` był wprowadzany do Pythona stopniowo: w 2.2 użycie go jako nazwy zmiennej generowało ostrzeżenia o przedawnieniu, a w 2.3 `yield` przestał być akceptowany jako nazwa i został wprowadzony jako słowo kluczowe.

Podobnie w Pythonie 2.6 nowymi słowniami kluczowymi stały się `with` i `as` używane przez menedżera kontekstu (służące między innymi do nowej formy obsługi wyjątków). Te dwa słowa kluczowe do wersji 2.5 nie były zarezerwowane, chyba że programista włączył obsługę menedżera kontekstów za pomocą instrukcji `from __future__ import` (co zostało omówione w dalszej części książki). W przypadku użycia `with` i `as` jako nazw zmiennych w 2.5 Python generuje ostrzeżenia o zbliżającej się zmianie. Wyjątkiem jest tu IDLE dla Pythona 2.5, w którym autorzy włączyli użycie tej funkcji (co w IDLE powoduje wyświetlenie błędu użycia słowa kluczowego, zamiast ostrzeżenia o przedawnieniu).

Konwencje dotyczące nazewnictwa

Oprócz powyższych reguł istnieje również zbiór *konwencji* dotyczących nazewnictwa — reguł, które nie są obowiązkowe, ale zazwyczaj się ich przestrzega. Na przykład ponieważ nazwy z dwoma znakami `_` na początku i końcu (jak `_name_`) zazwyczaj mają dla interpretera Pythona specjalne znaczenie, powinniśmy unikać tego wzorca we własnych nazwach zmiennych. Poniżej znajduje się lista konwencji dotyczących nazw zmiennych używanych w Pythonie.

- Nazwy rozpoczynające się od jednego znaku `_` (jak `_X`) nie są importowane za pomocą instrukcji `from moduł import *` (opisano to w rozdziale 23.).
- Nazwy z dwoma początkowymi i końcowymi znakami `_` (jak `_X_`) są nazwami zdefiniowanymi przez system, które mają specjalne znaczenie dla interpretera.
- Nazwy rozpoczynające się od dwóch znaków podkreślenia `_` i niekończące się dwoma kolejnymi takimi znakami (jak `_X_`) są lokalne dla zawierających je klas (opisano to w rozdziale 31.).
- Nazwa będąca pojedynczym znakiem podkreślenia `_` przechowuje w sesji interaktywnej wynik ostatniego wyrażenia.

Oprócz powyższych konwencji interpretera Pythona istnieją różne inne konwencje, których zazwyczaj przestrzegają programiści tego języka. W dalszej części książki zobaczymy na przykład, że nazwy klas najczęściej zaczynają się od wielkiej litery, a także przekonamy się, że nazwa `self` — choć nie jest zarezerwowana — zazwyczaj pełni w klasach pewną specjalną rolę. W rozdziale 17. zajmiemy się również inną, większą kategorią nazw zwanych *wbudowanymi*, które także są z góry zdefiniowane, jednak nie zarezerwowane (dlatego można je przypisać ponownie: `open = 42` zadziała, choć czasami wolelibyśmy, żeby tak nie było!).

Nazwy nie mają typu, ale obiekty tak

To przede wszystkim przypomnienie, jednak rozróżnianie nazw i obiektów Pythona jest kwestią kluczową. Jak pisaliśmy w rozdziale 6., obiekty mają typ (na przykład liczby całkowitej czy listy) i mogą być zmienne lub niezmienne. Nazwy (inaczej zmienne) są zawsze jedynie referencjami do obiektów. Nie jest z nimi powiązana koncepcja zmienności i niezmienności ani informacja o typie — poza typem obiektu, do którego w określonym momencie są referencją.

Można przypisać tę samą nazwę do różnych rodzajów obiektów w różnym czasie.

```
>>> x = 0                                # x powiązane z obiektem liczby
całkowitej

>>> x = "Troll"                            # Teraz jest łańcuchem znaków

>>> x = [1, 2, 3]                          # A teraz listą
```

W późniejszych przykładach zobaczymy, że uniwersalna natura zmiennych może być dużą zaletą programowania w Pythonie. W rozdziale 17. książki dowiemy się, że zmienne istnieją w czymś o nazwie *zakres*, co definiuje miejsca, w których mogą one być użyte. Miejsce, w którym przypisujemy zmienną, określa to, gdzie będzie ona widoczna^[4].



Dodatkowe sugestie dotyczące konwencji nazewnictwa możesz przeczytać w półoficjalnym przewodniku po języku Python, znanym jako PEP 8. Przewodnik ten jest dostępny na stronie <http://www.python.org/dev/peps/pep-0008>; możesz go też znaleźć, wpisując w wyszukiwarce sieciowej frazę *Python PEP 8*. Z technicznego punktu widzenia ten dokument formalizuje standardy tworzenia kodu biblioteki Pythona.

Sformalizowane standardy kodowania są bardzo użyteczne, ale warto traktować je z rezerwą. Po pierwsze, PEP 8 zawiera więcej szczegółów, niż większość Czytelników jest w stanie przyswoić sobie na tym etapie studiowania niniejszej książki. Szczerze mówiąc, reguły zdefiniowane w PEP 8 są zbyt

skomplikowane, sztywne i subiektywne, niż można by oczekiwąć od dokumentu tego typu. Niektóre sugestie w nim zawarte są wręcz powszechnie kwestionowane lub wręcz ignorowane przez programistów Pythona. Co więcej, wiele firm używających Pythona zaadaptowało własne standardy, które różnią się z PEP 8 w wielu kwestiach.

PEP 8 jednak stanowi ważny punkt odniesienia w zakresie standardu tworzenia kodu w Pythonie i jest doskonałą lekturą dla początkujących programistów Pythona, jednak należy traktować go jako zbiór zaleceń, a nie prawd objawionych.

Instrukcje wyrażeń

W Pythonie można również użyć wyrażenia jako instrukcji (to znaczy w osobnym wierszu). Ponieważ jednak wynik wyrażenia nie zostanie zapisany, ma to sens jedynie wtedy, gdy efekt uboczny działania wyrażenia będzie przydatny. Wyrażenia są często używane w charakterze instrukcji w dwóch sytuacjach.

W wywołaniach funkcji i metod

Niektóre funkcje oraz metody wykonują dużo pracy bez zwracania wartości. Takie funkcje czasami nazywane są w innych językach *procedurami*. Ponieważ nie zwracają wartości, które moglibyśmy chcieć zachować, możemy je wywoływać za pomocą instrukcji wyrażeń.

Do wyświetlania wartości w sesji interaktywnej

Python zwraca wyniki wyrażenia wpisanego w interaktywnym wierszu poleceń. Z technicznego punktu widzenia są one również instrukcjami wyrażeń — służą jako skrót zastępujący wpisywanie instrukcji `print`.

W tabeli 11.4 wymieniono często spotykane formy instrukcji wyrażeń w Pythonie. Wywołania funkcji oraz metod są kodowane z zerową lub większą liczbą argumentów (a tak naprawdę wyrażeń zwracających obiekty) w nawiasach, po nazwie funkcji lub metody.

Tabela 11.4. Często wykorzystywane instrukcje wyrażeń Pythona

Operacja	Interpretacja
<code>spam(eggs, ham)</code>	Wywołanie funkcji
<code>spam.ham(eggs)</code>	Wywołanie metody
<code>spam</code>	Wyświetlanie zmiennych w interpreterze interaktywnym
<code>print(a, b, c, sep='')</code>	Funkcja wypisywania ciągów znaków w Pythonie 3.x
<code>yield x ** 2</code>	Zwracanie wyniku częściowego

Dwa ostatnie wiersze w tabeli 11.4 mają specjalną formę; jak przekonamy się w dalszej części niniejszego rozdziału, wyświetlanie tekstów w Pythonie 3.x jest realizowane przez funkcję, natomiast wyrażenie `yield` w funkcjach generatorów (o tym dowiemy się więcej w rozdziale 20.) jest często zapisywane jak instrukcja. Jedno i drugie jest natomiast specyficzną formą instrukcji wyrażeń.

Na przykład funkcja `print` w Pythonie 3.x jest z reguły wywoływana w osobnym wierszu kodu bez przypisania wyniku, choć funkcja `print` zwraca wynik jak każda inna (a dokładniej zwraca

wartość `None`, co jest zupełnie normalne w przypadku funkcji, których zadanie nie polega na zwracaniu wyniku).

```
>>> x = print('spam')      # print jest w 3.x wyrażeniem wywołania funkcji
spam
>>> print(x)            # najczęściej jest jednak stosowana jak instrukcja
wyrażenia
None
```

Pamiętaj jednak, że choć wyrażenia mogą się w Pythonie pojawić w postaci instrukcji, instrukcje nie mogą być używane jako wyrażenia. Instrukcja, która nie jest wyrażeniem, musi na ogół występować sama w wierszu, a nie zagnieżdżona w większej strukturze składniowej. Python nie pozwala na przykład osadzać instrukcji przypisania (`=`) w innych wyrażeniach. Uzasadnienie jest takie, że pozwala to uniknąć błędów w kodowaniu. Nie możemy przypadkowo zmodyfikować zmiennej, wpisując `=`, kiedy tak naprawdę chciałibyśmy użyć testu równości `==`. Zobaczmy, jak obejść to ograniczenie, kiedy w rozdziale 13. spotkamy się z pętlą `while`.

Instrukcje wyrażeń i modyfikacje w miejscu

W ten sposób dochodzimy do błędu często popełnianego w pracy z Pythonem. Instrukcje wyrażeń są często wykorzystywane w celu wykonywania metod list modyfikujących te listy w miejscu.

```
>>> L = [1, 2]
>>> L.append(3)           # append jest metodą
modyfikującą listę w miejscu
>>> L
[1, 2, 3]
```

Osoby rozpoczynające swoją znajomość z Pythonem często kodują taką operację jako instrukcję przypisania, zamierzając przypisać `L` do większej listy.

```
>>> L = L.append(4)       # append zwraca None, a nie L
>>> print L              # Tracimy więc naszą listę!
None
```

To jednak nie działa — wywołanie operacji modyfikującej listę w miejscu, takiej jak `append`, `sort` czy `reverse`, zawsze zmienia tę listę w miejscu, jednak metody te nie zwracają samej zmodyfikowanej listy. Tak naprawdę zwracają one obiekt `None`. Jeśli przypiszemy wynik takiej operacji z powrotem do zmiennej, w efekcie całkowicie stracimy listę (i najprawdopodobniej zostaje ona w międzyczasie usunięta z pamięci przez mechanizm czyszczenia).

Morał z tej historii jest taki, że nie należy tego robić — operacje modyfikujące w miejscu powinny być zawsze wywoływanie bez przypisywania wyników. Z tym zjawiskiem spotkamy się raz jeszcze w sekcji „Często spotykane problemy programistyczne” znajdującej się na końcu tej części książki, ponieważ może się ono pojawić również w kontekście niektórych instrukcji pętli omawianych w kolejnych rozdziałach.

Polecenia `print`

Polecenia `print` wyświetlają (drukują) różne rzeczy — to po prostu przyjazny programiście interfejs do standardowego strumienia wyjścia.

Z technicznego punktu widzenia można powiedzieć, że przekształca on obiekt na jego reprezentację tekstową, dodaje formatowanie i przesyła do standardowego wyjścia. Nieco bardziej szczegółowo — `print` jest silnie związany z plikami i strumieniami wyjścia w Pythonie:

Metody plikowe

W rozdziale 9. można się było zapoznać z informacjami na temat obiektów plikowych służących do zapisu tekstu (`file.write(str)`). Wyświetlanie informacji działa podobnie, ale w sposób bardziej specjalizowany: metody zapisu do plików pozwalają zapisywać dane w dowolnych plikach, natomiast funkcja `print` wypisuje teksty na standardowym wyjściu z możliwością określenia formatowania. W przeciwieństwie do operacji plikowych, przy użyciu funkcji `print` nie ma konieczności przekształcania obiektów na ciągi znaków.

Standardowy strumień wyjściowy

Standardowy strumień wyjściowy (znany jako `stdout`) jest podstawowym mechanizmem systemowym do wypisywania wyników przez programy. Wraz ze standardowym strumieniem wejścia oraz strumieniem błędów jest jednym z trzech kanałów komunikacyjnych tworzonych podczas uruchamiania skryptu. Standardowy strumień wyjściowy jest z reguły mapowany na ekran (terminal), z którego został uruchomiony program, chyba że w poleceniu wywołania zostanie przekierowany do potoku lub pliku.

Standardowy strumień wyjściowy jest dostępny w Pythonie jako obiekt `stdout` wchodzący w skład modułu `sys` biblioteki standardowej (`sys.stdout`). Istnieje możliwość zaemulowania funkcji `print` z użyciem standardowych mechanizmów plikowych. Jednak funkcja `print` jest znaczco prostsza w użyciu, jak również pozwala zapisywać teksty w plikach i innych strumieniach.

Wyświetlanie tekstów to jedno z zagadnień, w których wystąpiły najwyraźniejsze różnice między Pythonem 3.x a 2.x. W rzeczywistości ta różnica między wersjami jest z reguły pierwszą przyczyną braku możliwości uruchomienia kodu 2.x w Pythonie 3.x. Sposób wywołania operacji wyświetlania tekstu różni się bowiem w różnych wersjach Pythona:

- w Pythonie 3.x do wyświetlania tekstu stosuje się *funkcję wbudowaną print*, obsługującą argumenty ze słowami kluczowymi pozwalające na zastosowanie specjalnych trybów wyświetlania;
- w Pythonie 2.x wyświetlanie tekstu jest instrukcją wykorzystującą własną, specyficzną składnię.

Ponieważ niniejsza książka opisuje zarówno Pythona 2.x, jak i 3.0, w kolejnych podrozdziałach omówimy obydwa te sposoby wyświetlania. Jeżeli tworzysz kod tylko w jednej z omawianych wersji Pythona, możesz z powodzeniem pominąć punkty, które dotyczą nieużywanej wersji. Ponieważ Twoje potrzeby mogą się jednak zmieniać, prawdopodobnie nie zaszkodzi Ci zaznajomienie się z obydwoma przypadkami. Co więcej, użytkownicy ostatnich wydań Pythona 2.x mogą również importować i wykorzystywać sposób wyświetlania z użyciem polecenia `print` z wersji 3.x w swoich programach, jeżeli jest to pożądane — zarówno ze względu na dodatkową funkcjonalność, jak i ułatwienie przyszłej migracji całego kodu do wersji 3.x.

Funkcja `print` z Pythona 3.x

Szczerze mówiąc, wyświetlanie tekstów w wersji 3.x nie wykorzystuje żadnej specjalnej instrukcji. Zamiast tego mamy do dyspozycji wyrażenie wykorzystujące *wywołanie funkcji*, wspomniane w poprzednim punkcie.

Wbudowana funkcja `print` jest najczęściej wywoływana w osobnym wierszu kodu bez przypisywania zwracanej wartości (ponieważ `print` zwraca `None`, tak jak to pokazywaliśmy w

poprzedniej sekcji). Z uwagi na to, że w wersji 3.x `print` jest funkcją, mamy do dyspozycji standardową składnię wywołania funkcji, a nie specjalną formę instrukcji jak w starszych wersjach języka. Funkcja `print` obsługuje różne tryby działania obsługiwane za pomocą argumentów ze słowami kluczowymi, dzięki czemu jej użycie jest bardziej uogólnione i pozwala na proste wprowadzanie nowych mechanizmów w przeszłości.

Dla porównania instrukcja `print` z wersji 2.x wykorzystuje specjalną składnię pozwalającą na pominięcie znaków nowego wiersza lub przekierowanie wyników działania do pliku. Co więcej, stara forma w ogóle nie pozwala na zdefiniowanie separatorów; w wersji 2.x znacznie częściej tworzymy predefiniowane ciągi znaków, niż ma to miejsce w 3.x. Zamiast rozbudowywania ad hoc jego składni wywołania polecenie `print` w Pythonie 3.x zostało mocno uogólnione, dzięki czemu to jedno polecenie obejmuje teraz wszystkie przypadki zastosowań.

Format wywołania

Składniowo wywołanie funkcji `print` w Pythonie 3.x ma następującą formę (w wersji 3.3 wprowadzono nowy argument `flush`):

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout][, flush=False])
```

W tej formalnej notacji elementy w nawiasach kwadratowych są opcjonalne i można je pominąć w wywołaniu, wartości po znaku = określają domyślne wartości argumentów. Mówiąc bardziej przystępnie, wbudowana funkcja `print` przesyła do pliku `file` (domyślnie standardowe wyjście) tekstową reprezentację jednego lub więcej obiektów oddzielonych separatorem `sep` i zakończonych znakiem `end`, opróżniając zbuforowane dane wyjściowe lub nie w zależności od wartości `flush`.

Argumenty `sep`, `end`, `file`, a w wersji 3.3 i nowszych także `flush` muszą być podane jako *argumenty ze słowami kluczowymi*, to znaczy w celu ich przekazania musi być użyta specjalna składnia nazwa=wartość, pozwalająca na przekazywanie argumentów według nazwy, a nie pozycji w wierszu wywołania. Argumenty ze słowami kluczowymi zostały omówione szczegółowo w rozdziale 18., a ich użycie jest proste. Argumenty tego typu przekazywane do funkcji mogą mieć dowolną kolejność w wierszu wywołania, a ich znaczenie jest następujące:

- `sep` określa ciąg znaków wstawiany pomiędzy wyświetlanymi obiektami, domyślnie to jedna spacja; w celu rezygnacji z separatorów należy przekazać pusty串 znaków.
- `end` określa串 znaków, jaki będzie dołączany na końcu wyświetlanego tekstu, domyślnie jest to sekwencja końca wiersza (\n). Przekazanie pustego串 znaków spowoduje, że na końcu wyświetlanego tekstu kursor nie zostanie przeniesiony do nowego wiersza, czyli następna operacja `print` rozpocznie się na końcu ostatnio wyświetlonego wiersza.
- `file` określa plik, strumień standardowy lub inny obiekt typu plikowego, do którego zostanie wysłany wyświetlany串 tekst. Domyślną wartością tego argumentu jest `sys.stdout`, czyli standardowy strumień wyjściowy. W tym argumencie można przekazać dowolny obiekt plikowy obsługujący metodę `write(ciąg_znaków)`, z tym że rzeczywiste pliki muszą być już otwarte do zapisu.
- `flush`, argument dodany w wersji 3.3, domyślnie ma wartość `False`; pozwala na natychmiastowe opróżnienie bufora do strumienia wyjściowego (najczęściej pliku). Zwykle to, czy wydruk jest buforowany w pamięci, czy nie, zależy od wartości argumentu `file`; przekazanie argumentu `flush=True` wymusza opróżnienie bufora do strumienia wyjściowego.

Tekstowa reprezentacja każdego obiektu jest tworzona przez przekazanie go do wbudowanej funkcji `str` (lub jej wewnętrznego odpowiednika w Pythonie). Jak widzieliśmy wcześniej, ta funkcja zwraca „przyjazną dla użytkownika” reprezentację tekstową każdego obiektu^[5]. W przypadku wywołania funkcji `print` bez argumentów funkcja wysyła do standardowego strumienia wyjściowego znak nowego wiersza, co zazwyczaj powoduje wyświetlenie pustego wiersza.

Funkcja `print` z wersji 3.x w działaniu

Wyświetlanie ciągów znaków w 3.x jest znacznie prostsze, niż może sugerować nagłówek funkcji `print`. Przeanalizujmy kilka prostych przykładów. Poniższy listing prezentuje wyświetlanie obiektów różnych typów z domyślnym separatorem i zakończeniem (te wartości są zastosowane jako domyślne, ponieważ są najczęściej stosowane).

```
C:\code> c:\python33\python
>>>
>>> print()                                     # Wyświetla pustą linię
>>> x = 'mielonka'
>>> y = 99
>>> z = ['jajka']
>>>
>>> print(x, y, z)                            # Wyświetla 3 obiekty z
domyślnym formatowaniem
mielonka 99 ['jajka']
```

Nie ma potrzeby przekształcania obiektów na ciągi znaków, co jest konieczne w przypadku zapisu obiektów do pliku. Domyślnie funkcja `print` rozdziela reprezentacje obiektów pojedynczą spacją. Aby tego uniknąć, należy przekazać pusty ciąg znaków w argumencie `sep`, można oczywiście podać własny separator.

```
>>> print(x, y, z, sep='')                      # Pominięcie separatora
mielonka99['jajka']
>>>
>>> print(x, y, z, sep=', ')                   # Niestandardowy separator
mielonka, 99, ['jajka']
```

Również domyślnie funkcja `print` na końcu dodaje znak nowego wiersza. I to zachowanie można zmienić, przekazując w argumencie `end` pusty ciąg znaków lub inną sekwencję znaków, która ma być wyświetlona na końcu (znak końca wiersza jest reprezentowany przez sekwencję `\n`). Drugie z poniższych poleceń to dwie instrukcje w jednym wierszu, oddzielone średnikiem:

```
>>> print(x, y, z, end='')                      # Pominięcie przejścia do
nowego wiersza
mielonka 99 ['jajka']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)      # Dwie instrukcje print
wyświetlają tekst w tym samym wierszu
mielonka 99 ['jajka']mielonka 99 ['jajka']
>>> print(x, y, z, end='...\n')                  # Niestandardowa sekwencja
zakończenia wiersza
mielonka 99 ['jajka']...
>>>
```

Można również zastosować różne kombinacje argumentów ze słowami kluczowymi, definiując własne separatory i zakończenia wierszy: argumenty mogą występować w dowolnej kolejności

po liście wyświetlanych obiektów.

```
>>> print(x, y, z, sep='...', end='!\n')      # Wiele argumentów ze
słowami kluczowymi
mielonka...99...['jajka']!
>>> print(x, y, z, end='!\n', sep='...')        # Kolejność nie ma
znaczenia
mielonka...99...['jajka']!
```

Poniższy listing prezentuje użycie argumentu `file`. W przypadku przekazania otwartego obiektu plikowego obiekty zostaną zapisane w tym pliku, zamiast być wyświetlane na ekranie; taka zamiana obowiązuje tylko w tym wywołaniu funkcji `print` (to jedna z form przekierowania strumienia wyjściowego, tym zagadnieniem zajmiemy się szerzej za chwilę).

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'))    # Wypisanie do
pliku
>>> print(x, y, z)                                              # Powrót do
standardowego wyjścia
mielonka 99 ['jajka']
>>> print(open('data.txt').read())                                # Wyświetlenie
pliku tekstowego
mielonka...99...['jajka']
```

Należy pamiętać, że argumenty `sep` i `end` są dostępne jedynie dla wygody. Jeżeli chcemy zastosować bardziej skomplikowane formatowanie, można zupełnie z nich zrezygnować i ręcznie budować ciągi znaków, korzystając z zaawansowanego formatowania i wykorzystując narzędzia poznane w rozdziale 7., a następnie wyświetlać takie gotowe ciągi znaków w jednym wywołaniu funkcji `print`.

```
>>> text = '%s: %-.4f, %05d' % ('Wynik', 3.14159, 42)
>>> print(text)
Wynik: 3.1416, 00042
>>> print('%s: %-.4f, %05d' % ('Wynik', 3.14159, 42))
Wynik: 3.1416, 00042
```

Jak się przekonamy w następnym punkcie, prawie wszystko, co wiemy o funkcji `print` z Pythonem 3.x, ma zastosowanie do instrukcji `print` z Pythona 2.x, co ma sens, biorąc pod uwagę fakt, że funkcja ma za zadanie emulować stary mechanizm, uzupełniając go o nowe możliwości.

Instrukcja `print` w Pythonie 2.x

Jak wspomniałem wcześniej, do wyświetlania ciągów znaków w Pythonie 2.x wykorzystywana jest instrukcja `print` z własną, specjalizowaną składnią. W praktyce jednak wyświetlanie tekstu w 2.x jest wariacją poznanego mechanizmu wyświetlania z Pythona 3.x, z pominięciem możliwości definiowania separatorów (które są dostępne w 3.x, ale niedostępne w 2.x) oraz opróżniania bufora wyjściowego (dostępne od wersji 3.3). Oprócz tego wszystko, co da się zrobić za pomocą funkcji `print` w 3.x, można przekształcić w instrukcję `print` Pythona 2.x.

Formy instrukcji

Tabela 11.5 przedstawia formy wywołania instrukcji `print` w Pythonie 2.x oraz ich odpowiedniki w funkcji `print` Pythona 3.x. Warto zwrócić uwagę na to, że *przecinek* ma znaczenie w instrukcji `print`: służy do oddzielania wyświetlanych obiektów. Przecinek na końcu instrukcji zapobiega wyświetleniu znaku końca wiersza na końcu (nie należy mylić takiego zapisu ze składnią krotki!). Operator `>>` (normalnie używany jako przesunięcie bitowe w prawo) w kontekście instrukcji `print` jest używany do przekierowania wyniku do innego strumienia niż domyślny `sys.stdout`.

Tabela 11.5. Formy wywołania instrukcji `print` w Pythonie 2.x

Instrukcja w Pythonie 2.x	Odpowiednik w Pythonie 3.x	Interpretacja
<code>print x, y</code>	<code>print(x, y)</code>	Wypisuje na standardowym wyjściu <code>sys.stdout</code> reprezentacje tekstowe obiektów, oddzielając je spacją i umieszczając znak końca wiersza na końcu wyniku.
<code>print x,</code> <code>y,</code>	<code>print(x, y, end='')</code>	To samo, bez znaku końca wiersza.
<code>print >> afile, x, y</code>	<code>print(x, y, file=afile)</code>	Przekierowanie ciągu znaków do pliku zamiast do <code>sys.stdout</code> .

Instrukcja `print` Pythona 2.x w działaniu

Choć w wersji 2.x instrukcja `print` posiada bardziej unikatową składnię niż funkcja `print` w 3.x, jej użycie jest bardzo podobne. Ponownie przeanalizujmy kilka prostych przykładów. Instrukcja `print` w 2.x dodaje spację między wyświetlonymi obiektami oddzielonymi przecinkami, a na końcu domyślnie dodaje znak nowego wiersza.

```
C:\code> c:\python27\python
>>> x = 'a'
>>> y = 'b'
>>> print x, y
a b
```

To formatowanie jest domyślne i można z niego korzystać lub nie. Aby uniknąć dodawania znaku nowego wiersza na końcu wyświetlonego tekstu, należy instrukcję zakończyć przecinkiem, jak w drugim wierszu tabeli 11.5 (na poniższym listingu dwie instrukcje `print` zostały umieszczone w jednym wierszu, oddzielone średnikiem).

```
>>> print x, y; print x, y
a b a b
```

Aby uniknąć spacji między obiektami, nie należy oddzielać obiektów przecinkami. Zamiast tego można wynikowy ciąg znaków zbudować w całości przed wyświetleniem, używając narzędzi formatowania ciągów znaków poznanych w rozdziale 7., po czym wyświetlić utworzony ciąg znaków.

```
>>> print x + y
ab
```

```
>>> print '%s...%s' % (x, y)
a...b
```

Jak widzimy — pomijając specjalną składnię — instrukcja `print` Pythona 2.x jest tak prosta w użyciu, jak funkcja `print` z Pythona 3.x. Następny punkt wyjaśni, w jaki sposób można wymusić zapis do pliku w Pythonie 2.x, co jest odpowiednikiem argumentu `file` funkcji `print`.

Przekierowanie strumienia wyjściowego

Zarówno w Pythonie 3.x, jak i 2.x użycie funkcji lub instrukcji `print` powoduje wyświetlenie ciągu znaków na ekranie. Często jednak użytkownicy mogą okazać się wysłanie wyniku w inne miejsce: na przykład do pliku tekstowego, w celu przeanalizowania wyników po zakończeniu działania programu. Choć tego typu przekierowanie wyniku do pliku można zrealizować poza Pythonem dzięki narzędziom powłoki systemowej, to czasem konieczne bywa zastosowanie kontroli strumienia wyjściowego bezpośrednio w skrypcie w Pythonie.

Program „Witaj, świecie!”

Zacznijmy zatem od tradycyjnego, powszechnie znanego (i zupełnie bezużytecznego) programu „Witaj, świecie!”. Oto wersje wywołań dla różnych wersji Pythona:

```
>>> print('Witaj, świecie!')                                # Wyświetlenie ciągu znaków w
Pythonie 3.x

Witaj, świecie!

>>> print 'Witaj, świecie!'                               # Wyświetlenie ciągu znaków w
Pythonie 2.x

Witaj, świecie!
```

Ponieważ wyniki wyrażeń są automatycznie wyświetlane w interaktywnym wierszu poleceń, często nie musisz nawet używać instrukcji `print` — po prostu wpisz wyrażenie, którego wartość chcesz zobaczyć, a jego wynik zostanie wyświetlony na ekranie:

```
>>> 'Witaj, świecie!'                                     # Interaktywne wyświetlanie
wartości

Witaj, świecie!
```

Powyższy przykład raczej nie stanowi zachwycającego przykładu mistrzowsko przygotowanego kodu, ale dobrze ilustruje zasadę działania mechanizmu wyświetlania. W rzeczywistości operacja `print` jest jedynie ergonomiczną wersją zapisu do pliku z prostym mechanizmem formatującym. Jeżeli lubisz pracować bardziej ciężko, niż to konieczne, możesz również zakodować wyświetlanie, właśnie korzystając z operacji plikowych (zgodnie z tym, co pisaliśmy w rozdziałach 4. i 9., wartość reprezentująca rozmiar ciągu, wyświetlana tylko w wersji 3.x, została tutaj pominięta).

```
>>> import sys                                         # Wyświetlanie metodą "plikową"

>>> sys.stdout.write('Witaj, świecie\n')

witaj świecie
```

Powyższy kod wywołuje metodę `write` obiektu plikowego `sys.stdout`, czyli obiektu podłączonego do standardowego strumienia wyjściowego inicjalizowanego podczas uruchamiania interpretera Pythona. Operacja `print` działa w zbliżony sposób, ale ukrywa przed programistą te szczegóły, dając mu proste narzędzie do realizacji prostych zadań związanych z wyświetlaniem tekstów.

Ręczne przekierowanie strumienia wyjścia

Po co zatem pokazałem trudniejszy sposób wyświetlania obiektów? Użycie `sys.stdout` okazuje się podstawą pewnej często stosowanej w Pythonie techniki. Generalnie `print` i `sys.stdout` powiązane są ze sobą w następujący sposób. Poniższa instrukcja:

```
print(X, Y)                                # albo w Pythonie 2.x: print X, Y
```

jest odpowiednikiem dłuższej sekwencji:

```
import sys  
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

która ręcznie przeprowadza konwersję łańcucha znaków za pomocą `str`, dodaje nowy wiersz za pomocą `+` i wywołuje metodę `write` strumienia wyjścia. Której wersji wolałbyś użyć? Piszę to, mając nadzieję na podkreślenie przyjaznego dla programisty charakteru funkcji `print`...

Oczywiście dłuższa forma nie jest szczególnie użyteczna dla zwykłego wyświetlania. Dobrze jest jednak wiedzieć, że instrukcje `print` robią dokładnie to, ponieważ można przypisać `sys.stdout` do czegoś innego niż standardowy strumień wyjścia. Innymi słowy, równoważność tych dwóch rozwiązań umożliwia tworzenie własnych instrukcji `print` przesyłających tekst do innych miejsc, tak jak pokazano w poniższym przykładzie.

```
import sys  
sys.stdout = open('log.txt', 'a')           # Przekierowuje print do pliku  
  
...  
print x, y, x                               # Zapisuje tekst w pliku log.txt
```

W tym przykładzie przekierowujemy `sys.stdout` do ręcznie otwieranego pliku o nazwie `log.txt`, znajdującego się w bieżącym katalogu roboczym i otwieranego w trybie dołączania danych (które będą dodawane do jego bieżącej zawartości). Po takiej zmianie każda instrukcja `print` w dowolnym miejscu programu będzie zapisywała tekst na końcu pliku `log.txt`, a nie w oryginalnym strumieniu wyjścia. Instrukcje `print` wywołują po prostu metodę `write` obiektu `sys.stdout` bez względu na to, do czego w danej chwili odnosi się `sys.stdout`. Ponieważ istnieje tylko jeden moduł `sys`, przypisanie `sys.stdout` w ten sposób przekieruje każdą instrukcję `print` znajdująjącą się w dowolnym miejscu programu.

Tak naprawdę, co zostanie wyjaśnione w ramce „Warto pamiętać — `print` i `stdout`”, możemy nawet ustawić `sys.stdout` na obiekt niebędący plikiem, o ile obsługuje on oczekiwany protokół (metodę `write`). Kiedy ten obiekt jest *klasą*, wyświetlany tekst można dowolnie przekierowywać i przetwarzać za pomocą własnoręcznie napisanej metody `write`.

Sztuczka z przekierowaniem strumienia wyjścia jest szczególnie użyteczna w programach oryginalnie napisanych z instrukcjami `print`. Jeżeli od początku wiemy, że dane wyjściowe powinny trafiać do pliku, zawsze możemy wywołać zamiast tego metody zapisu do pliku. Aby jednak przekierować dane wyjściowe programu opartego na instrukcjach `print`, nie musząc jednocześnie modyfikować każdej z tych instrukcji lub używać składni powłoki systemowej, wygodnie jest skorzystać z możliwości przekierowania `sys.stdout` lub przekierować strumień danych wyjściowych za pomocą poleceń na poziomie powłoki systemu operacyjnego.

W innych zastosowaniach strumienie danych mogą być przekierowywane do obiektów, które będą je wyświetlać w oknach graficznego interfejsu użytkownika i kolorować w zintegrowanych środowiskach programistycznych, takich jak IDLE i tak dalej — jest to technika ogólnego przeznaczenia.

Automatyczne przekierowanie strumienia

Sztuczka z przekierowaniem zapisywanego tekstu za pomocą przypisania `sys.stdout` jest w praktyce używana bardzo często. Jednym z problemów związanych z poprzednim kodem jest jednak to, że nie istnieje bezpośrednia metoda przywrócenia oryginalnego strumienia wyjścia, gdybyśmy musieli przełączyć się z powrotem po zapisaniu tekstu do pliku. Ponieważ `sys.stdout` jest normalnym obiektem pliku, zawsze możemy go jednak zapisać i w razie konieczności przywrócić^[6].

```
C:\code> c:\python33\python
>>> import sys
>>> temp = sys.stdout                      # Zapisanie na później
>>> sys.stdout = open('log.txt', 'a')        # Przekierowanie do pliku
>>> print('mielonka')                      # Wypisuje do pliku, nie na ekran
>>> print(1, 2, 3)
>>> sys.stdout.close()                     # Opróżnienie bufora zapisu
>>> sys.stdout = temp                      # Przywrócenie oryginalnego
strumienia
>>> print('znów jestem')                  # Wynik znów pojawia się na
ekranie
znów jestem
>>> print(open('log.txt').read())          # Rezultat wcześniejszego
wypisywania
mielonka
1 2 3
```

Jak pokazano w powyższym przykładzie, ręczne zapisywanie i przywracanie oryginalnego strumienia wyjścia w ten sposób wymaga sporej ilości dodatkowej pracy. Ponieważ jednak taka sytuacja zdarza się dość często, w instrukcji `print` zaimplementowano taką możliwość.

W wersji 3.x do tego celu służy argument `file` funkcji `print`, który pozwala na zapisanie tekstu do wskazanego pliku bez konieczności uprzedniego modyfikowania przypisania strumienia `sys.stdout`. Dzięki temu, że takie przekierowanie jest tymczasowe, kolejne wywołania funkcji `print` nadal będą wysyłyły tekst do oryginalnego strumienia wyjściowego. W wersji 2.x takie samo działanie ma instrukcja `print`, w której zastosuje się operator przekierowania `>>`. Poniższy przykład zapisuje tekst do pliku `log.txt`.

```
log = open('log.txt', 'a')                 # 3.x
print(x, y, z, file=log)                  # Zapis do pliku
print(a, b, c)                           # Zapis do oryginalnego strumienia
stdout
log = open('log.txt', 'a')                 # 2.x
print >> log, x, y, z                   # Zapis do pliku
print a, b, c                            # Zapis do oryginalnego strumienia
stdout
```

Tego typu przekierowania są szczególnie użyteczne w sytuacjach, gdy w tym samym kodzie musimy zapisywać dane do pliku i na ekran naprzemiennie. Jeżeli używasz takiego rozwiązania, powinieneś pamiętać, aby używać obiektu pliku (lub obiektu posiadającego taką samą metodę

`write`, co obiekt pliku), a nie ciągu znaków określającego nazwę pliku. Poniższy listing prezentuje tę technikę w działaniu:

```
C:\code> c:\python33\python
>>> log = open('log.txt', 'w')
>>> print(1, 2, 3, file=log)           # w wersji 2.x: print >> log, 1, 2, 3
>>> print(4, 5, 6, file=log)
>>> log.close()
>>> print(7, 8, 9)                   # w wersji 2.x: print 7, 8, 9
7 8 9
>>> print(open('log.txt').read())
1 2 3
4 5 6
```

Ta rozszerzona forma instrukcji `print` jest również często wykorzystywana do zapisywania komunikatów o błędach do standardowego strumienia wyjścia błędów, `sys.stderr`. Można albo skorzystać z metod zapisu pliku i ręcznie sformatować dane wyjściowe, albo użyć instrukcji `print` ze składnią przekierowującą.

```
>>> import sys
>>> sys.stderr.write('Źle!' * 8 + '\n')
Źle!Źle!Źle!Źle!Źle!Źle!Źle!
>>> print ('Źle!' * 8, file=sys.stderr,)    # w wersji 2.x: print >>
sys.stderr, 'Źle!' * 8
Źle!Źle!Źle!Źle!Źle!Źle!Źle!
```

Po poznaniu tajników wyświetlania znaków powiązanie tej operacji z metodą `write` pliku powinno być już oczywiste. Poniższy listing prezentuje sesję konsoli interaktywnej Pythona 3.x, w której wyświetlamy tekst przy użyciu obu metod (zwykłej i plikowej), po czym wynik zostaje przekierowany do pliku zewnętrznego w celu weryfikacji prawidłowości.

```
>>> X = 1; Y = 2
>>> print(X, Y)                      # Wyświetlanie:
metoda standardowa
1 2
>>> import sys                        # Wyświetlanie:
metoda plikowa
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
1 2
4
>>> print(X, Y, file=open('temp1', 'w'))          # Przekierowanie
do pliku
>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n') # Bezpośredni
zapis do pliku
```

```
>>> print(open('temp1', 'rb').read()) # Binarny tryb odczytu
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'
```

Jak widać, funkcja lub instrukcja `print` jest z reguły najlepszym sposobem wyświetlania tekstów, chyba że lubisz wpisywać duże ilości nadmiarowego kodu. Aby zobaczyć inny przykład równoważności funkcji `print` i rozwiązania z użyciem metody `print` plików, poszukaj w rozdziale 18. przykładu emulacji funkcji `print` z wersji 3.x, która używa tego wzorca kodu do zaimplementowania odpowiednika funkcji `print` z wersji 3.x do użytku w Pythonie 2.x.

Wyświetlanie niezależne od wersji

Jeżeli chcesz, aby wyświetlanie tekstów działało poprawnie w *obu* liniach Pythona, masz do wyboru kilka opcji. Dzieje się tak niezależnie od tego, czy piszesz kod 2.x, który ma być kompatybilny z wersją 3.x, czy też kod 3.x, który ma obsługiwać również wersję 2.x.

Konwerter 2to3

Po pierwsze, można użyć instrukcji `print` Pythona 2.x i użyć skryptu `2to3`, który automatycznie przekształci skrypty do postaci zgodnej z Pythonem 3.x.Więcej szczegółów na temat tego narzędzia można znaleźć w dokumentacji Pythona 3.x. Mówiąc w skrócie, skrypt `2to3` próbuje zmodyfikować kod napisany dla wersji 2.x w taki sposób, aby działał w wersjach 3.x — jest to bardzo przydatne narzędzie, ale czasami zbyt mocno próbuje sprawić, aby operacje drukowania były neutralne dla wersji. Powiązane z nim narzędzie o nazwie `3to2` próbuje wykonać operację odwrotną: przekonwertować kod z wersji 3.x tak, aby działał na wersji 2.x; więcej szczegółowych informacji na ten temat znajdziesz w załączniku C.

Importowanie z `_future_`

Alternatywnym rozwiązaniem może być zakodowanie wywołania funkcji `print` z wersji 3.x w kodzie uruchamianym przez wersję 2.x poprzez włączenie wariantu wywołania funkcji za pomocą instrukcji podobnej do przedstawionej poniżej. Instrukcja taka powinna być umieszczone na początku skryptu lub w dowolnym miejscu sesji interaktywnej:

```
from __future__ import print_function
```

Ta instrukcja zmienia funkcję `print` z wersji 2.x tak, aby działała identycznie jak funkcja `print` w wersji 3.x. W ten sposób możesz korzystać z funkcji `print` z wersji 3.x i nie będziesz musiał zmieniać sposobu wyświetlania, jeżeli później przeprowadzisz migrację do wersji 3.x. Dwie uwagi dotyczące użytkowania tego rozwiązania:

- Taka instrukcja jest po prostu *ignorowana*, jeżeli pojawia się w kodzie uruchomionym przez wersję 3.x — nie ma znaczenia, jeżeli zostanie zawarta w kodzie 3.x mającym być zgodnym z wersją 2.x.
- Taka instrukcja musi pojawiać się na początku *każdego pliku skryptu* w wersji 2.x, który będzie wykorzystywał mechanizmy wyświetlania z 3.x — dzieje się tak, ponieważ użycie tej instrukcji modyfikuje parser tylko dla bieżącego pliku, zatem nie wystarczy zaimportować do niego innego pliku niezawierającego tej instrukcji.

Neutralizacja różnic w wyświetlaniu za pomocą kodu

Pamiętaj, że proste instrukcje wyświetlania, jak prezentowane w tabeli 11.5, będą działać w dowolnej wersji Pythona, a ponieważ każde wyrażenie może być ujęte w nawiasy, zawsze możemy udawać, że wywołujemy funkcję `print` z wersji 3.x w wersji 2.x poprzez dodanie zewnętrznych nawiasów. Główną wadą takiego rozwiązania jest to, że w przypadku próby wyświetlenia kilku obiektów ujęcie ich w nawiasy przekształca je w *krotkę*, a więc na ekranie pojawią się dodatkowe nawiasy okrągłe. Na przykład w wersji 3.x w nawiasach wywołania funkcji `print` może znajdować się dowolna liczba obiektów:

```
C:\code> c:\python33\python
>>> print('mielonka')                                # Wywołanie funkcji print w
wersji 3.x
mielonka
>>> print('mielonka', 'szynka', 'jajka')      # użycie kilku argumentów
mielonka szynka jajka
```

Pierwsze wywołanie zadziała tak samo, jak w Pythonie 2.x, ale drugie wywołanie wygeneruje krotkę, która następnie zostanie wyświetlona:

```
C:\code> c:\python27\python
>>> print('mielonka')                                # Instrukcja print w 2.6 z
użyciem nawiasów
mielonka
>>> print('mielonka', 'szynka', 'jajka')      # To w rzeczywistości obiekt
krotki!
('mielonka', 'szynka', 'jajka')
```

To samo dotyczy sytuacji, gdy nie zostaną wyświetlane *żadne* obiekty wymuszające wysunięcie wiersza: wersja 2.x wyświetla krotkę, chyba że wydrukujesz pusty ciąg znaków:

```
c:\code> py -2
>> print()                                         # W wersji 3.x to tylko
wysunięcie wiersza
()
>>> print('')                                     # To wysunięcie wiersza zarówno w
wersji 2.x, jak i 3.x
```

Ściśle mówiąc, wyniki mogą w niektórych przypadkach różnić się czymś więcej niż tylko dodatkowymi nawiasami otaczającymi w wersji 2.x. Jeżeli przyjrzesz się uważnie powyższym wynikom, zauważysz, że ciągi są wyświetlane wraz z cudzysłowami tylko w wersji 2.x. Wynika to z faktu, że po zagnieżdżeniu w innym obiekcie obiekty mogą być wyświetlane inaczej niż jako elementy najwyższego poziomu. Z technicznego punktu widzenia elementy zagnieżdżone są wyświetlane z `repr`, a obiekty najwyższego poziomu ze `str` — są to dwa alternatywne formaty wyświetlania, o których wspominaliśmy w rozdziale 5.

Oznacza to po prostu dodatkowy cudzysłowy wokół ciągów znaków zagnieżdżonych w krotce zawierającej wiele elementów umieszczonych w nawiasach w wersji 2.x. Wyświetlanie zagnieżdżonych obiektów może się znacznie różnić w przypadku innych typów obiektów, a zwłaszcza obiektów klasy, które definiują alternatywne mechanizmy wyświetlania z wykorzystaniem *przeciążania operatorów* — omawianie tego zagadnienia rozpoczęliśmy w części VI, a w szczególności w rozdziale 30.

Aby kod był w pełni przenośny bez konieczności włączania wszędzie zgodności wyświetlanego z wersją 3.x i aby uniknąć różnic w wyświetaniu zagnieżdżonych elementów, zawsze możesz sformatować ciąg wydruku jako pojedynczy obiekt, ujednolicając tym samym wyświetlanie w różnych wersjach. Możesz to zrobić, używając wyrażenia formatującego ciąg, wywołania metody format lub innego narzędzia do formatowania łańcuchów, które omawialiśmy w rozdziale 7.:

```
>>> print('%s %s %s' % ('mielonka', 'szynka', 'jajka'))
mielonka szynka jajka
>>> print('{0} {1} {2}'.format('mielonka', 'szynka', 'jajka'))
mielonka szynka jajka
>>> print('odpowiedź: ' + str(42))
answer: 42
```

Oczywiście jeżeli możemy pozwolić sobie na używanie wyłącznie wersji 3.x, można zapomnieć o tych komplikacjach, ale wielu programistów Pythona może napotkać kod napisany w Pythonie 2.x, warto więc wiedzieć, jak sobie z nim radzić. W wielu przykładach w tej książce wykorzystamy zarówno `_future_`, jak i kod neutralny dla wersji, aby osiągnąć możliwość przenoszenia kodu między wersjami 2.x i 3.x.

Warto pamiętać — `print` i `stdout`

Równoważność instrukcji `print` i zapisu do `sys.stdout` ma duże znaczenie. Umożliwia przypisanie `sys.stdout` do obiektu zdefiniowanego przez użytkownika udostępniającego te same metody co pliki (na przykład `write`). Ponieważ instrukcja `print` po prostu przesyła tekst do metody `sys.stdout.write`, możemy przechwycić tekst zapisywany w naszych programach, przypisując `sys.stdout` do obiektu, którego metoda `write` w dowolny sposób przetworzy ten tekst.

Na przykład można przesłać wyświetlany tekst do okna graficznego interfejsu użytkownika lub wysłać go do kilku miejsc docelowych, definiując obiekt z metodą `write` wymagającą przekierowania. Przykład takiej sztuczki zobaczymy przy okazji omawiania klas w szóstej części książki, jednak w skrócie będzie on przypominał poniższe rozwiązanie.

```
class FileFaker:
    def write(self, string):
        # Coś robi z łańcuchem znaków
    import sys
    sys.stdout = FileFaker()
    print(someObjects)                                # Przesyła do metody write
    klasy
```

Takie rozwiązanie działa, ponieważ `print` jest czymś, co w następnej części książki nazwiemy operacją *polimorficzną* — dla instrukcji tej nie ma znaczenia, czym jest `sys.stdout`, ważne jest tylko, by obiekt ten miał metodę (a właściwie interfejs) `write`. W nowszych wersjach Pythona takie przekierowanie do obiektów jest jeszcze łatwiejsze dzięki rozszerzonej formie `print` ze znakami `>>` — nie musimy już w jawnym sposobie przestawiać `sys.stdout`.

```
myobj = FileFaker()                                # 3.x: Przekierowanie do obiektu
na tę jedną operację
print(someObjects, file= myobj)                    # Nie przestawia sys.stdout
```

```
myobj = FileFaker()                                # 2.x: taki sam efekt
print >> myobj, someObjects                         # Nie przestawia sys.stdout
```

W Pythonie 3.x wbudowana funkcja `input` (w wersji 2.x nosi ona nazwę `raw_input`) odczytuje dane ze standardowego wejścia (pliku) `sys.stdin`, dzięki czemu w podobny sposób można przechwytywać żądania odczytu, wykorzystując do tego klasy implementujące metody `read` podobne do tych używanych w obiektach plików.Więcej informacji na ten temat można znaleźć w przykładzie z `input` i pętlą `while` z rozdziału 10.

Warto zauważyć, że ponieważ wyświetlany tekst trafia do strumienia `stdout`, w ten sposób można wyświetlać strony HTML odpowiedzi w skryptach CGI używanych w internecie. Pozwala nam to również na przekierowanie danych wejściowych i wyjściowych skryptu Pythona w systemowym wierszu poleceń w normalny sposób:

```
python script.py < plik_wejsciowy > plik_wyjsciowy
```

```
python script.py | filterProgram
```

Narzędzia przekierowania strumieni wbudowane w funkcję i instrukcję `print` Pythona to w zasadzie pythonowa implementacja tych narzędzi powłoki systemowej.Więcej informacji na temat skryptów CGI i składni poleceń powłoki możesz znaleźć w innych źródłach (np. w internecie).



W książce w wielu miejscach wykorzystuję wywołanie funkcji `print` z Pythona 3.x. Zazwyczaj staram się tak dobierać polecenia, aby wyświetlanie tekstu na ekranie było niezależne od wersji Pythona i z reguły uprzedzam Czytelnika w przypadku, gdy wynik danej operacji może wyglądać nieco inaczej w wersji 2.x. Czasami jednak tak nie jest, zatem proszę, potraktuj tę uwagę jako ogólne ostrzeżenie. Jeżeli uruchomisz przykładowy kod w Pythonie 2.x i zobaczysz na ekranie nawiasy, których nie ma w książce, spróbuj usunąć nawiasy z instrukcji `print`, zaimportować funkcję `print` w wersji 3.x z modułu `__future__`, przepisać polecenia zgodnie z opisanymi wyżej regułami tworzenia kodu wyświetlającego dane w sposób niezależny od wersji Pythona lub po prostu spróbuj polubić taki nadmiarowy tekst pojawiający się na ekranie.

Podsumowanie rozdziału

W niniejszym rozdziale rozpoczęliśmy nasze pogłębione omówienie instrukcji Pythona od zapoznania się z przypisaniem, wyrażeniami i instrukcjami `print`. Choć wszystkie one są stosunkowo proste w użyciu, mają pewne opcjonalne formy alternatywne, które w praktyce często się przydają. Rozszerzone instrukcje przypisania i forma instrukcji `print` z przekierowaniem pozwalają na przykład uniknąć ręcznego kodowania pewnych kwestii. Przy okazji omówiliśmy składnię nazw zmiennych, techniki przekierowywania strumienia wyjścia i wiele często popełnianych błędów, takich jak przypisanie wyniku wywołania metody `append` z powrotem do zmiennej.

W kolejnym rozdziale będziemy kontynuować omawianie instrukcji, uzupełniając brakujące szczegóły dotyczące instrukcji `if` — najważniejszego narzędzia służącego do dokonywania wyboru w kodzie napisanym w Pythonie. Powrócimy również do modelu składni Pythona i przyjrzymy się zachowaniu wyrażeń Boolean. Zanim jednak do tego dojdzie, quiz podsumowujący rozdział pozwoli sprawdzić wiedzę nabycą przy okazji lektury tekstu.

Sprawdź swoją wiedzę — quiz

1. Podaj trzy sposoby przypisania tej samej wartości do trzech zmiennych.
2. Dlaczego warto zachować ostrożność, przypisując trzy zmienne do obiektu mutowalnego?
3. Co złego jest w kodzie `L = L.sort()`?
4. W jaki sposób można wykorzystać instrukcję `print` do przesłania tekstu do pliku zewnętrznego?

Sprawdź swoją wiedzę — odpowiedzi

1. Można skorzystać z przypisania z wieloma celami (`A = B = C = 0`), przypisania sekwencji (`A, B, C = 0, 0, 0`) lub kilku osobnych instrukcji przypisania w odrębnych wierszach (`A = 0, B = 0, C = 0`). W przypadku tej ostatniej techniki (zgodnie z informacjami z rozdziału 10.) można również połączyć trzy odrębne instrukcje w jeden wiersz, rozdzielając je średnikami (`A = 0; B = 0; C = 0`).

2. Jeżeli przypiszemy je w poniższy sposób:

```
A = B = C = []
```

wszystkie trzy zmienne będą się odnosiły do tego samego obiektu, dlatego modyfikacja jednej z nich w miejscu (na przykład `A.append(99)`) będzie miała wpływ na pozostałe dwie zmienne. Ma to znaczenie wyłącznie w przypadku modyfikacji obiektów mutowalnych (jak listy i słowniki) w miejscu; w przypadku obiektów niemutowalnych, takich jak liczby i łańcuchy znaków, kwestia ta nie ma znaczenia.

3. Metoda listy `sort` jest podobna do metody `append`, ponieważ modyfikuje listę będącą jej podmiotem w miejscu — zwraca obiekt `None`, a nie zmodyfikowaną listę. Przypisanie jej wyniku z powrotem do `L` daje ustawienie `L` na `None`, a nie posortowaną listę. Jak wspominaliśmy już wcześniej (np. w rozdziale 8.) i jak zobaczymy niebawem w dalszej części książki, nowsza, wbudowana funkcja `sorted` sortuje dowolną sekwencję i zwraca nową, posortowaną listę; ponieważ nie jest ona modyfikacją w miejscu, jej wynik może być przypisany z powrotem do zmiennej.
4. Można przypisać `sys.stdout` do ręcznie otwartego pliku przed użyciem instrukcji `print` lub skorzystać z rozszerzonej formy instrukcji `print >> file` w celu zapisania tekstu do pliku w tej jednej instrukcji. Można również przekierować całość wyświetlanego tekstu programu do pliku za pomocą specjalnej składni polecenia powłoki systemowej, jednak wykracza to poza kwestie związane z Pythonem.

[1] Uwaga dla programistów języków C i C++: choć Python obsługuje teraz instrukcje takie, jak `X += Y`, nadal nie ma operatorów automatycznej inkrementacji i dekrementacji z języka C (czyli `X++` i `--X`). Nie do końca odpowiadają one modelowi obiektów Pythona, ponieważ Python nie pozwala na modyfikacje w miejscu obiektów niemutowalnych, takich jak liczby.

[2] Zgodnie z sugestią z rozdziału 6., można również wykorzystać przypisanie do wycinków (na przykład `L[:len(L)] = [11, 12, 13]`), jednak działa to mniej więcej tak samo jak prostsza i bardziej mnemoniczna metoda `extend` listy.

[3] Dzieje się tak przynajmniej w standardowej implementacji CPython. Alternatywne implementacje języka Python mogą pozwalać, aby nazwy zmiennych zdefiniowanych przez użytkownika były takie same jak słowa zastrzeżone w języku Python. Przegląd alternatywnych implementacji, takich jak Jython, znajdziesz w rozdziale 2.

[4] Osoby używające w przeszłości języka C++ zainteresować może to, że w Pythonie nie istnieje koncepcja deklaracji `const` z tego języka. Pewne obiekty mogą być *niemutowalne*, jednak nazwy można przypisywać zawsze. Python potrafi również ukrywać nazwy w klasach i modułach, jednak nie jest to to samo co deklaracja z C++ (jeżeli ukrywanie atrybutów jest dla Ciebie ważne, omówienie nazw typu `_X` w modułach znajdziesz w rozdziale 25., opis nazw typu `_X` w klasach w rozdziale 31., a zagadnień związanych z dekoratorami prywatnych i publicznych atrybutów klas w rozdziale 39.).

[5] Technicznie rzecz biorąc, funkcja `print` wykorzystuje wewnętrzny odpowiednik funkcji `str`, ale efekt jest dokładnie taki sam. Funkcja `str` nie tylko zwraca tekstową reprezentację obiektów, ale też jest stosowana jako funkcja przekształcająca dane do typu tekstowego i może być użyta do dekodowania ciągów znaków Unicode z użyciem dodatkowego argumentu określającego sposób kodowania.Więcej informacji na ten temat znajdziesz w rozdziale 37.; ten ostatni sposób użycia jest jednak bardzo zaawansowanym zagadnieniem i możemy go bezpiecznie zignorować w kontekście bieżącego rozdziału.

[6] W obu wersjach Pythona, 2.x oraz 3.x, można również skorzystać z względnie nowego atrybutu `__stdout__` modułu `sys`, który odnosi się do oryginalnej wartości `sys.stdout` z początku programu. Aby jednak wrócić do oryginalnej wartości strumienia wyjścia, trzeba będzie przywrócić `sys.stdout` do `sys.__stdout__`.Więcej informacji na ten temat można znaleźć w dokumentacji modułu `sys`.

Rozdział 12. Testy if i reguły składni

Niniejszy rozdział wprowadza instrukcję `if`, będącą w Pythonie podstawowym narzędziem wykorzystywanym w wyborze alternatywnych działań w oparciu o wyniki testu. Ponieważ będzie to nasze pierwsze szersze omówienie *instrukcji złożonych* — czyli instrukcji z osadzonymi innymi instrukcjami — bardziej szczegółowo niż w rozdziale 10. zajmiemy się również uniwersalnymi koncepcjami stojącymi za modelem składni instrukcji Pythona. Ponieważ instrukcje warunkowe wprowadzają pojęcie testów, rozdział ten będzie również omawiał wyrażenia typu Boolean i trójelementową instrukcję `if` oraz uzupełni pominięte dotychczas informacje dotyczące testów prawdy.

Instrukcje if

W uproszczeniu instrukcja `if` Pythona wybiera działanie, które należy wykonać. To podstawowe narzędzie wyboru w tym języku, reprezentujące dużą część *logiki* programu napisanego w Pythonie. Podobnie do innych instrukcji złożonych Pythona, może ona zawierać inne instrukcje, w tym kolejne `if`. Tak naprawdę Python pozwala nam łączyć instrukcje w programie w sposób sekwencyjny (tak by były wykonywane jedna po drugiej) i dowolnie zagnieżdżony, tak, aby wykonywane były jedynie pod pewnymi warunkami, takimi jak wybory i pętle.

Ogólny format

Instrukcja `if` w Pythonie jest typowym przykładem instrukcji warunkowych `if` występujących w większości języków proceduralnych. Przybiera formę testu `if`, po którym następuje jeden lub większa liczba testów `elif` (od „`else if`”) oraz końcowy opcjonalny blok `else`. Testy oraz część `else` zawierają powiązane bloki zagnieżdżonych instrukcji, wciętych w stosunku do wiersza nagłówka. Kiedy instrukcja `if` jest wykonywana, Python wykonuje blok kodu powiązany z pierwszym testem zwracającym wynik będący prawdą lub blok `else`, jeżeli wszystkie testy zwracają wynik będący fałszem. Ogólna forma instrukcji `if` przedstawia się w następujący sposób:

```
if test1:                                # Test if
    instrukcje1                            # Powiązany blok
elif test2:                                # Opcjonalne testy elif
    instrukcje2
else:                                     # Opcjonalna klauzula else
    instrukcje3
```

Proste przykłady

Aby zademonstrować działanie instrukcji `if`, przyjrzyjmy się kilku krótkim przykładom. Wszystkie części tej instrukcji są opcjonalne, za wyjątkiem początkowego testu `if` oraz powiązanych z nim poleceń; tym samym w najprostszym przypadku pozostałe części są pomijane.

```
>>> if 1:  
...     print('prawda')  
...  
prawda
```

Warto zwrócić uwagę, jak znak zachęty zmienia się na `...` w wierszach kontynuacji podstawowego interfejsu wykorzystywanego w sesji interaktywnej (w IDLE zamiast tego przechodzimy po prostu do kolejnego, wciętego wiersza — by powrócić do początku wiersza, należy nacisnąć klawisz *Backspace*). Pusty wiersz (który uzyskuje się za pomocą dwukrotnego naciśnięcia klawisza *Enter*) kończy i wykonuje całą instrukcję. Należy pamiętać, że `1` to inaczej wartość logiczna oznaczająca prawdę (jak się przekonasz niebawem, jej odpowiednikiem jest słowo `True`), dlatego warunek powyższej instrukcji zawsze będzie spełniony. Aby obsłużyć wynik będący fałszem, powinieneś utworzyć klauzulę `else`.

```
>>> if not 1:  
...     print('prawda')  
... else:  
...     print('fałsz')  
...  
fałsz
```

Rozgałęzienia kodu

Poniżej znajduje się przykład bardziej złożonej instrukcji `if`, w której obecne są wszystkie opcjonalne części.

```
>>> x = 'zabójczy królik'  
>>> if x == 'roger':  
...     print("gdzie jest Jessie?")  
... elif x == 'bugs':  
...     print("co słychać, doktorku?")  
... else:  
...     print('Uciekaj! Uciekaj!')  
...  
Uciekaj! Uciekaj!
```

Ta wielowierszowa instrukcja rozciąga się od wiersza z `if` aż do bloku `else`. Po uruchomieniu Python wykonuje instrukcje zagnieździone pod pierwszym testem zwracającym prawdę lub pod częścią `else`, jeśli wszystkie z testów zwracają fałsz (tak, jak w powyższym przykładzie). W praktyce zarówno `elif`, jak i `else` można pominąć, a w każdej z osadzonych części może być więcej instrukcji. Warto zwrócić uwagę na to, że słowa `if`, `elif` i `else` są ze sobą powiązane za pomocą wcięć — są wyrównane w pionie.

Osoby znające języki programowania takie, jak C lub Pascal, może zainteresować fakt, iż w Pythonie nie ma instrukcji `switch` czy `case` wybierającej działanie w oparciu o wartość zmiennej. Zamiast tego *rozgałęzienia kodu* są zazwyczaj zapisywane albo jako seria testów `if` i `elif`, jak w przykładzie wyżej, albo za pomocą indeksowania słowników lub przeszukiwania list. Ponieważ słowniki i listy mogą być tworzone dynamicznie po uruchomieniu programu, czasami są bardziej elastyczne od zakodowanej na stałe logiki `if`.

```
>>> choice = 'szynka'

>>> print({'mielonka': 1.25,                      # Instrukcja 'switch' oparta na
          słowniku
          ...
          'szynka': 1.99,                            # Wartość domyślna dzięki
          has_key lub get
          ...
          'jajka': 0.99,
          ...
          'boczek': 1.10}[choice])

1.99
```

Choć zrozumienie takiego rozwiązania może przy pierwszej styczności z nim zająć chwilę, słownik ten jest tak naprawdę rozgałęzieniem kodu. Indeksowanie po kluczu `choice` rozgałęzia kod na jedną wartość ze zbioru, podobnie jak robi to instrukcja `switch` z języka C. Prawie równoważne, acz nieco bardziej rozwlekłe rozwiązanie z wykorzystaniem instrukcji `if` może z kolei wyglądać jak kod przedstawiony poniżej:

```
>>> if choice == 'mielonka':                  # odpowiednik rozgałęzienia kodu
       z użyciem instrukcji if
       ...
       print(1.25)
...
elif choice == 'szynka':
...
print(1.99)
...
elif choice == 'jajka':
...
print(0.99)
...
elif choice == 'boczek':
...
print(1.10)
...
else:
...
print('Zły wybór')

...
1.99
```

Wprawdzie taki kod może być bardziej czytelny, niemniej potencjalnym minusem tego rozwiązania jest to, że choć można utworzyć go w postaci łańcucha i uruchomić za pomocą narzędzi takich jak `eval` lub `exec` z poprzedniego rozdziału, to nie można go tak łatwo budować dynamicznie po uruchomieniu programu jak w przypadku słownika. W bardziej dynamicznych programach struktury danych oferują dodatkową elastyczność.

Obsługa domyślnych wartości wyboru

Zwróć uwagę na klauzulę `else` polecenia `if`, która obsługuje domyślny przypadek, gdy żaden klucz nie pasuje. Jak widzieliśmy w rozdziale 8., wartości domyślne słownika można zakodować za pomocą wyrażeń, wywołań metod `get` lub przechwytywania wyjątków za pomocą instrukcji `try` wprowadzonej w poprzednim rozdziale. Wszystkie wspomniane techniki mogą być tutaj

użyte do zakodowania domyślnej akcji w rozgałęzionym kodzie opartym na słowniku. Poniżej przedstawiamy przykład realizacji takiego zadania przy użyciu metody `get`.

```
>>> branch = {'mielonka': 1.25,
...             'szynka': 1.99,
...             'jajka': 0.99}
>>> print(branch.get('mielonka', 'Zły wybór'))
1.25
>>> print(branch.get('boczek', 'Zły wybór'))
Zły wybór
```

Test przynależności `in` umieszczony w instrukcji `if` może mieć ten sam domyślny efekt.

```
>>> choice = 'boczek'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Zły wybór')
...
Zły wybór
```

Warto zauważyć, że instrukcja `try` jest ogólnym sposobem obsługiwanego wartości domyślnych poprzez przechwytywanie i obsługę wyjątków, które w przeciwnym wypadku zostałyby zgłoszone (więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 11. oraz w części VII):

```
>>> try:
...     print(branch[choice])
... except KeyError:
...     print('Zły wybór')
...
Zły wybór
```

Obsługa bardziej złożonych operacji

Słowniki dobrze nadają się do łączenia wartości z kluczami, jednak co z bardziej skomplikowanymi działaniami, jakie możemy zapisać w blokach instrukcji powiązanych z instrukcjami `if`? W czwartej części książki zobaczymy, że słowniki mogą również zawierać *funkcje* reprezentujące bardziej skomplikowane działania i implementujące uniwersalne tablice skoków (ang. *jump table*). Takie funkcje pojawiają się jako wartości słownika, mogą być tworzone w postaci nazw funkcji lub funkcji śródziemowych `lambda` i wywoływane są przez dodanie nawiasów rozpoczętym ich działaniem. Poniżej zamieszczały małą, abstrakcyjną próbkę takiego rozwiązania, ale w rozdziale 19. wróćmy do tego zagadnienia po bardziej szczegółowym omówieniu definicji funkcji:

```
def function(): ...
def default(): ...
```

```

branch = {'mielonka': lambda: ...,
          # Tabela wywoływalnych
          # obiektów funkcji
          'szynka': function,
          'jajka': lambda: ...}

branch.get(choice, default)()

```

Choć rozgałęzianie oparte na słownikach przydaje się w programach obsługujących bardziej dynamiczne dane, większość programistów zauważą zapewne, że najprostszym sposobem wykonania rozgałęzienia kodu jest skorzystanie z instrukcji `if`. Uniwersalna reguła dotycząca programowania mówi, że kiedy mamy wątpliwości, lepiej jest postawić na prostotę i czytelność — będzie to najbardziej „pythonowy” wybór.

Reguły składni Pythona raz jeszcze

Model składni Pythona został wprowadzony w rozdziale 10. Teraz, gdy przechodzimy do większych instrukcji, takich jak `if`, czas na przypomnienie i rozszerzenie przedstawionych wcześniej koncepcji związanych ze składnią. Python ma prostą składnię opartą na instrukcjach. Istnieje jednak kilka jej właściwości, o których należy wiedzieć.

- **Instrukcje wykonywane są jedna po drugiej, o ile nie wskażemy innego sposobu.** Python normalnie wykonuje instrukcje w pliku lub osadzonym bloku w kolejności od pierwszej do ostatniej jako *sekwencję poleceń*, jednak instrukcje takie jak `if` (i jak zobaczymy — również pętle i wyjątki) sprawiają, że interpreter może przeskakiwać do innych miejsc w kodzie. Ponieważ „ścieżka”, po której Python przechodzi przez kolejne polecenia, wykonując program, nazywana jest *przebiegiem sterowania* (ang. *control flow*), instrukcje takie jak `if`, wpływające na ten przebieg, nazywane są *instrukcjami sterującymi przebiegiem programu* (ang. *control-flow statements*).
- **Granice bloków i instrukcji wykrywane są w sposób automatyczny.** Jak widzieliśmy, wokół bloku kodu napisanego w Pythonie nie ma nawiasów klamrowych czy innych ograniczników typu `begin/end`. Zamiast tego Python wykorzystuje wcięcia instrukcji pod nagłówkiem do pogrupowania instrukcji w zagnieżdżony blok. Analogicznie instrukcje Pythona nie są kończone średnikami; zamiast tego koniec wiersza oznacza zazwyczaj koniec instrukcji umieszczonej w tym wierszu. W szczególnym przypadku instrukcje mogą rozciągać się na wiele wierszy łączonych w jedno polecenie z użyciem specjalnej składni.
- **Instrukcje złożone składają się z wiersza nagłówka, znaku dwukropka i bloku wciętych instrukcji.** Wszystkie *instrukcje złożone* w Pythonie tworzone są zgodnie z tym samym wzorcem. Najpierw występuje wiersz nagłówka zakończony dwukropkiem, po nim jedna lub większa liczba zagnieżdżonych instrukcji, zazwyczaj wciętych pod nagłówkiem. Wcięte instrukcje nazywane są *blokiem*. W instrukcji `if` klauzule `elif` i `else` są jej elementami, ale jednocześnie również wierszami nagłówka dla własnych zagnieżdżonych bloków. W szczególnym przypadku bloki kodu mogą pojawiać się w tym samym wierszu co nagłówki, jeżeli mają postać prostych poleceń.
- **Puste wiersze, spacje i komentarze są zazwyczaj ignorowane.** Puste wiersze są opcjonalne i ignorowane w plikach (ale już nie w sesji interaktywnej, gdzie oznaczają koniec instrukcji złożonej). Spacje wewnętrzne instrukcji i wyrażeń są prawie zawsze ignorowane (z wyjątkiem literałów łańcuchów znaków i kiedy są używane do utworzenia wcięć kodu). Komentarze ignorowane są zawsze — zaczynają się od znaku `#` (nie wewnętrz literała łańcucha znaków) i rozciągają aż do końca bieżącego wiersza.
- **Notki dokumentacyjne są ignorowane, ale zapisywane i wyświetlane przez różne narzędzia.** Python obsługuje dodatkową formę komentarzy znaną jako *notki dokumentacyjne* (lub w skrócie *notki docstrings*), które — w przeciwieństwie do komentarzy ze znakiem `#` — są zachowywane po uruchomieniu programu. Są to po prostu

specjalne łańcuchy znaków umieszczane na początku plików programów i niektórych instrukcji. Python ignoruje ich zawartość, jednak są one automatycznie dołączane do obiektów w czasie wykonywania i mogą być wyświetlane za pomocą narzędzi do dokumentacji, takich jak PyDoc. Notki dokumentacyjne są częścią większej strategii dokumentacji Pythona i zostaną omówione w ostatnim rozdziale tej części książki.

Jak już się przekonałeś, w Pythonie nie używamy deklaracji typu zmiennych. Już ten fakt sprawia, że składnia tego języka jest o wiele prostsza niż to, do czego możemy być przyzwyczajeni. Dla większości nowych użytkowników brak nawiasów klamrowych i średników oznaczających w wielu innych językach programowania bloki i instrukcje wydaje się jednak największą nowością składniową w Pythonie, dlatego spróbujmy wyjaśnić tę kwestię nieco bardziej szczegółowo.

Ograniczniki bloków — reguły tworzenia wcięć

Jak wspominaliśmy już w rozdziale 10., Python automatycznie wykrywa granice bloków kodu dzięki *wcięciom* wierszy — czyli pustej przestrzeni po lewej stronie kodu. Wszystkie instrukcje wcięte na tę samą odległość w prawą stronę należą do tego samego bloku kodu. Innymi słowy, instrukcje jednego bloku są ze sobą wyrównane w pionie, jak kolumna. Blok kończy się na końcu pliku lub po napotkaniu mniej wciętego wiersza. Bloki głębiej zagnieżdżone są po prostu wcinane na większą odległość w prawą stronę w stosunku do instrukcji z bloku je zawierającego. Zawartość wyrażeń złożonych może czasami pojawiać się w wierszu nagłówka (będziemy o tym mówić później), ale w większości przypadków znajduje się pod nim i jest wyróżniona odpowiednim wcięciem.

Na rysunku 12.1 pokazano przykładową strukturę bloków kodu.

```
x = 1
if x:
    y = 2
    if y:
        print('blok2')
    print('blok1')
print('blok0')
```



Rysunek 12.1. Zagnieździone bloki kodu: blok zagnieżdżony rozpoczyna się od instrukcji wciętej na większą odległość w prawo i kończy na instrukcji wciętej na mniejszą odległość lub na końcu pliku

Powyższy kod zawiera trzy bloki. Pierwszy z nich (najwyższy poziom pliku) nie jest w ogóle wcięty, drugi (wewnętrz zewnętrznej instrukcji `if`) jest wcięty o trzy spacje, a trzeci (instrukcja `print` pod zagnieżdżonym `if`) wcięty jest o sześć spacji.

Zazwyczaj kod najwyższego poziomu (niezagnieżdżony) musi rozpoczynać się w pierwszej kolumnie. Bloki zagnieżdżone mogą zaczynać się w dowolnej kolumnie. Wcięcie może zawierać dowolną liczbę spacji i tabulatorów, dopóki dla wszystkich instrukcji w jednym bloku wcięcie będzie takie samo. Pythona nie interesuje, *w jaki sposób* będziemy wcinać kod. Ważne jest jedynie to, by odbywało się to w sposób spójny. Popularnymi konwencjami są cztery spacje lub jeden tabulator na jeden poziom wcięcia, jednak w świecie Pythona nie ma jednolitego standardu w tym zakresie.

Tworzenie wcięć kodu jest w praktyce bardzo naturalne. Przykładowo poniższy (dość niemądry) fragment kodu ilustruje często spotykane błędy tworzenia wcięć w kodzie napisanym w Pythonie.

```

x = 'MIELONKA'                                # Błąd – wcięcie pierwszego
wiersza

if 'plot' in 'żywopłot':
    print(x * 8)
    x += 'NI'                                    # Błąd – nieoczekiwane wcięcie

    if x.endswith('NI'):
        x *= 2
    print(x)                                     # Błąd – niespójne wcięcia

```

Wersję tego kodu z poprawnymi wcięciami pokazano poniżej, a nawet w przypadku tak naciąganego przykładu poprawne wcięcia kodu sprawiają, że jego zawartość jest o wiele bardziej zrozumiała:

```
x = 'MIELONKA'
```

```

if 'plot' in 'żywopłot':
    print(x * 8)                                # Wyświetla 8 razy słowo
MIELONKA

x += 'NI'

if x.endswith('NI'):
    x *= 2

print(x)                                      # Wyświetla
"MIELONKANIMIELONKANI"

```

Jednym istotnym miejscem w Pythonie, w którym białe znaki mają znaczenie, jest użycie ich po lewej stronie kodu — do tworzenia wcięć. We wszystkich innych kontekstach spacji można użyć lub nie. Wcięcia są tak naprawdę częścią składni Pythona, a nie tylko wskazówką stylistyczną. Wszystkie instrukcje w jednym bloku muszą być wcięte na tę samą odległość, gdyż inaczej Python zwróci nam błąd składni. Jest to celowe — ponieważ nie musimy w jawnym sposobie oznaczać końca zagnieżdzonego bloku kodu, część zbędnych elementów składniowych występująca w innych językach programowania mogła zostać w Pythonie pominięta.

Jak wspominaliśmy w rozdziale 10., sam fakt, że wcięcia stały się częścią modelu składni, powoduje wymuszenie spójności kodu — kluczowy element czytelności w ustrukturyzowanych językach programowania, takich jak Python. Składnia Pythona czasami opisywana jest jako „to, co widzisz, jest tym, co otrzymujesz” (ang. *WYSIWYG* — *What You See Is What You Get*) — wcięcie każdego wiersza kodu jednoznacznie informuje użytkownika, z czym kod ten jest powiązany. Ten spójny wygląd sprawia, że kod napisany w Pythonie łatwiej jest utrzymywać i wykorzystywać powtórnie.

Tworzenie wcięć jest o wiele bardziej naturalne, niż mogłyby to sugerować podane tu informacje, a dzięki nim kod programu lepiej odzwierciedla swoją strukturę logiczną. Spójne wcinanie kodu zawsze spełnia wymagania składni Pythona. Co więcej, większość edytorów tekstu (włącznie ze środowiskiem IDLE) ułatwia przestrzeganie reguł modelu składni Pythona poprzez automatyczne tworzenie wcięć kodu w miarę pisania.

Unikaj mieszania tabulatorów i spacji — nowa opcja sprawdzania błędów w Pythonie 3.x

Zasada numer jeden: choć do tworzenia wcięć kodu można wykorzystać spacje lub tabulatory, *mieszanie* obu rodzajów białych znaków w jednym bloku kodu nie jest najlepszym pomysłem. Lepiej jest konsekwentnie używać albo jednego, albo drugiego rozwiązania. Technicznie rzecz biorąc, domyślne tabulatory zawierają wystarczającą liczbę spacji, by przesunąć początek kodu o wielokrotność ośmiu znaków spacji; kod będzie działał poprawnie, jeżeli tabulatory i spacje będziemy ze sobą łączyć w spójny sposób. Taki kod może jednak być trudniejszy do modyfikacji. Co gorsza, pomieszanie ze sobą spacji i tabulatorów sprawia, że kod będzie znacznie trudniejszy do przeglądania i analizowania — tabulatory mogą wyglądać zupełnie inaczej w edytorach tekstu innych programistów.

Z tego właśnie powodu Python 3.x zwraca błąd, jeżeli w skrypcie tabulatory i spacje wykorzystane do tworzenia wcięć kodu użyte są w ramach jednego bloku w sposób niespójny (to znaczy w sposób uzależniony od ekwiwalentu spacji w tabulatorach). Python 2.x pozwala na uruchamianie takich skryptów, jednak udostępnia także w wierszu polecień opcję `-t`, która ostrzega przed niespójnym użyciem tabulatorów, oraz opcję `-tt`, zwracającą błędy dla takiego kodu (przełączniki te można zastosować podczas uruchamiania programu z poziomu wiersza polecień powłoki systemowej, na przykład: `python -t main.py`). Komunikaty o błędach w Pythonie 3.x wyświetlane w takiej sytuacji są odpowiednikiem przełącznika `-tt` z Pythona 2.x.

Ograniczniki instrukcji — wiersze i znaki kontynuacji

Normalnie instrukcja w Pythonie kończy się z końcem wiersza, w którym się znajduje. Kiedy instrukcja jest zbyt dłuża, aby zmieścić się w jednym wierszu, można skorzystać z kilku specjalnych reguł umożliwiających rozciągnięcie jej na kilka wierszy.

- **Instrukcje mogą rozciągać się na kilka wierszy, jeżeli kontynuujemy otwartą parę znaków składniowych.** Python pozwala na kontynuację instrukcji w kolejnym wierszu, jeżeli kod umieszczony jest w parze nawiasów zwykłych (()), klamrowych ({ }) lub kwadratowych ([]). Wyrażenia w nawiasach czy literały słowników i list mogą się rozciągać na dowolną liczbę wierszy; instrukcja nie kończy się, dopóki interpreter Pythona nie dojdzie do końca wiersza, w którym wpisałeś zamkającą część nawiasów (znak), } lub]). Wiersze kontynuacji (czyli wiersze od drugiego i dalej) mogą rozpoczynać się od dowolnego poziomu zagnieżdżenia, jednak z uwagi na czytelność powinieneś wyrównywać je w pionie. Reguła dotycząca otwierania par nawiasów dotyczy także zbiorów i słowników składanych z Pythona 3.x i 2.7.
- **Instrukcje mogą rozciągać się na kilka wierszy, jeżeli kończą się znakiem lewego ukośnika (\).** To nieco przestarzała i generalnie mocno już niezalecana opcja, ale jeśli instrukcja ma się rozciągać na kilka wierszy, można również dodać znak ukośnika lewego (znak \ nieosadzony w literale łańcucha znaków lub komentarzu) na końcu poprzedniego wiersza, aby wskazać, że będzie on kontynuowany w następnym. Ponieważ można osiągnąć to samo za pomocą pary nawiasów umieszczonej wokół większości konstrukcji, ukośniki lewe w zasadzie nie są już stosowane. Takie rozwiązanie jest podatne na błędy — przypadkowe pominięcie znaku ukośnika generuje błąd składni i może nawet powodować, że kolejny wiersz zostanie po cichu błędnie wzięty za nową instrukcję, co może dawać dosyć nieoczekiwane rezultaty.
- **Specjalne reguły dotyczące literałów łańcuchów znaków.** Jak wiemy z rozdziału 7., bloki łańcuchów znaków opatrzone potrójnymi apostrofami lub cudzysłowami normalnie służą do rozciągania długich łańcuchów znaków na kilka wierszy. Z rozdziału tego wiemy również, że przylegające do siebie literały łańcuchów znaków są w sposób niejawny poddawane konkatenacji. W połączeniu ze wspomnianą wcześniej regułą dotyczącą otwartych par nawiasów opakowanie takiej konstrukcji w nawiasy pozwala na rozciągnięcie łańcucha znaków na kilka wierszy.
- **Inne reguły.** Istnieje jeszcze kilka kwestii związanych z ogranicznikami instrukcji. Choć jest to stosunkowo rzadkie, instrukcję możemy kończyć średnikiem — taki zapis pozwala na umieszczenie większej liczby prostych (niezłożonych) instrukcji w jednym wierszu. Również komentarze i puste wiersze mogą pojawiać się w dowolnym miejscu pliku. Komentarze (rozpoczynające się od znaku #) kończą się na końcu wiersza, w którym występują.

Kilka przypadków specjalnych

Poniżej widać, jak może wyglądać wiersz kontynuacji w przypadku zastosowania reguły składniowej z otwartymi parami nawiasów. Konstrukcje oznaczone w ten sposób, takie jak listy umieszczone w nawiasach kwadratowych, mogą się rozciągać na dowolną liczbę wierszy.

```
L = ["Dobry",
      "Zły",
      "Paskudny"] # Para nawiasów obejmuje kilka
                  wierszy
```

Takie rozwiązanie będzie także działało dla dowolnej zawartości nawiasów zwykłych (wyrażeń, argumentów funkcji, nagłówków funkcji, krotek i wyrażeń generatorów), a także zawartości nawiasów klamrowych (słowników, a w Pythonie 3.x i 2.7 również literałów zbiorów oraz słowników składanych). Część tych narzędzi omówimy w kolejnych rozdziałach, jednak reguła ta w sposób naturalny obejmuje większość konstrukcji, które w praktyce rozciągają się na kilka wierszy.

Do kontynuowania wiersza można również użyć znaków lewego ukośnika, ale nie jest to w Pythonie często spotykane.

```
if a == b and c == d and \
   d == e and f == g:
    print('przestarzałe')      # Ukośniki lewe pozwalają na
kontynuację
```

Ponieważ każde wyrażenie może być ujęte w nawiasy, jeżeli chcesz rozciągnąć kod na wiele wierszy, możesz użyć rozwiązań z parą nawiasów — po prostu umieść część instrukcji w nawiasach:

```
if (a == b and c == d and
    d == e and e == f):
    print('nowy')             # To samo robią nawiasy (i są bardziej
oczywistym rozwiązaniem!)
```

Tak naprawdę używanie znaków lewego ukośnika jest mocno krytykowane przez wielu deweloperów, ponieważ takie znaki zbyt łatwo można przeoczyć lub nawet całkowicie je pominąć. W poniższym kodzie po użyciu znaku ukośnika do zmiennej `x` przypisana zostaje wartość 10, zgodnie z zamierzeniami. Jeżeli jednak znak `\` „przypadkowo” pominiemy, do zmiennej `x` przypisana zostanie zamiast tego wartość 6 i nie zostanie zgłoszony żaden błąd (+4 jest samo w sobie poprawnym wyrażeniem).

W prawdziwym programie z bardziej skomplikowaną operacją przypisania mogłoby to być źródłem wyjątkowo nieprzyjemnych błędów^[1].

```
x = 1 + 2 + 3 \
zmienia kod
+4
```

W innym przypadku specjalnym Python pozwala na zapis więcej niż jednej prostej instrukcji (instrukcji bez zagnieżdżonych innych instrukcji) w tym samym wierszu, rozdzielonych od siebie średnikami. Niektórzy programiści wykorzystują tę formę do zaoszczędzenia miejsca w pliku programu, jednak zazwyczaj kod jest czytelniejszy, kiedy przestrzegamy reguły umieszczania jednej instrukcji w wierszu.

```
x = 1; y = 2; print(x)          # Więcej niż jedna prosta instrukcja
```

Jak wiemy z rozdziału 7., literały tekstowe ujęte w potrójne znaki apostrofów także mogą się rozciągać na kilka wierszy. Dodatkowo jeżeli dwa literały łańcuchów znaków pojawiają się obok siebie, zostają połączone w procesie konkatenacji, tak jakby pomiędzy nie wstawiony został znak `+`. W połączeniu z regułą otwartych par nawiasów oznacza to, że ujęcie w nawiasy pozwala na rozciąganie tej formy zapisu na kilka wierszy. Pierwszy z poniższych przykładów kodu wstawia znaki nowego wiersza w miejscu złamania wiersza i przypisuje ciąg '`\naaaa\nbbbb\ncccc`' do zmiennej `S`, natomiast drugi przykład dokonuje niejawniej konkatenacji i przypisuje tej zmiennej ciąg '`aaaabbbbcccc`'. W drugim przykładzie komentarze są ignorowane, ale w pierwszym są dołączane do łańcucha znaków:

```
S = """
aaaa
bbbb
cccc"""
S = ('aaaa'
```

```
'bbbb' # Tutaj komentarze są ignorowane  
'cccc')
```

Wreszcie Python pozwala również na przeniesienie ciała instrukcji złożonej do wiersza nagłówka, pod warunkiem że ciało jest pojedynczą, prostą instrukcją. Najczęściej można to zobaczyć w prostych instrukcjach `if` z pojedynczym testem i działaniem, jakich używaliśmy na przykład w pętlach w rozdziale 10.:

```
if 1: print('witaj') # Prosta instrukcja w wierszu  
nagłówka
```

Niektóre z tych przypadków specjalnych można ze sobą łączyć i w rezultacie otrzymać kod trudny do odczytania, jednak nie polecam takiego działania. Powinieneś trzymać się reguły jednej instrukcji na wiersz i wcinać wszystkie bloki kodu z wyjątkiem tych najprostszych. Po sześciu miesiącach programowania będziesz szczęśliwy, że wyrobiliś sobie taki nawyk.

Testy prawdziwości i testy logiczne

Pojęcia porównania, równości oraz wartości prawdy zostały wprowadzone w rozdziale 9. Ponieważ instrukcja `if` jest pierwszą omawianą instrukcją, która tak naprawdę wykorzystuje wynik testu, rozszerzymy nieco koncepcje przedstawione wcześniej. Operatory logiczne Pythona (operatory typu Boolean) nieznacznie się różnią od swoich odpowiedników w językach takich, jak C. W Pythonie wygląda to tak:

- Wszystkie obiekty mają z natury przypisane wartości logiczne reprezentujące prawdę lub fałsz.
- Dowolna liczba niebędąca zerem i dowolny niepusty obiekt są prawdą.
- Liczby o wartości zero, puste obiekty i specjalny obiekt `None` uznawane są za fałsz.
- Porównania i testy równości stosowane są do struktur danych w sposób rekurencyjny.
- Porównania i testy równości zwracają `True` i `False` (odpowiedniki liczb 1 i 0).
- Operatory logiczne `and` i `or` zwracają obiekt reprezentujący prawdę lub fałsz.
- Operacje wykorzystujące operatory logiczne kończą działanie od razu, gdy tylko znany jest wynik wyrażenia (tzw. analiza skrócona).

Instrukcja `if` sprawdza, czy określony warunek został spełniony, ale operatory typu Boolean są używane również do tworzenia bardziej rozbudowanych wyrażeń i sprawdzania ich wartości logicznych. Mówiąc bardziej ogólnie, w Pythonie wyróżniamy trzy operatory wyrażeń logicznych:

`X and Y`

Wyrażenie jest prawdziwe, kiedy zarówno `X`, jak i `Y` są prawdziwe.

`X or Y`

Wyrażenie jest prawdziwe, kiedy `X` lub `Y` jest prawdziwe.

`not X`

Wyrażenie jest prawdziwe, kiedy `X` jest fałszywe (wyrażenie zwraca `True` lub `False`).

`X` i `Y` mogą mieć dowolną wartość logiczną lub mogą być dowolnym wyrażeniem zwracającym wartość logiczną (na przykład testem równości lub porównaniem zakresu). Operatory logiczne wpisywane są w Pythonie jako słowa (zamiast zapisu `&&`, `||` i `!` z języka C). Operatory `and` oraz `or` zwracają w Pythonie obiekty reprezentujące prawdę lub fałsz, a nie wartości `True` czy `False`. Przyjrzyjmy się kilku przykładom, aby przekonać się, jak to działa.

```
>>> 2 < 3, 3 < 2 # Mniejszy od – zwraca True lub
False (1 lub 0)
(True, False)
```

Porównania wielkości, jak powyższy przykład, zwracają `True` i `False` — wartości, które (jak wiemy z rozdziałów 5. i 9.) są tylko inaczej zapisanymi wersjami liczb całkowitych `1` oraz `0`. Są one jedynie wyświetlane w inny sposób, jednak poza tym niczym się nie różnią.

Z drugiej strony, operatory `and` oraz `or` zawsze zwracają obiekt — albo obiekt znajdujący się po lewej stronie operatora, albo ten z prawej strony. Jeżeli sprawdzimy ich wartości w `if` lub innych instrukcjach, będą one zgodne z oczekiwaniemi (należy pamiętać, że każdy obiekt jest z natury prawdą, albo fałszem), jednak nie otrzymamy z powrotem prostych wartości `True` lub `False`.

W przypadku operatora `or` Python analizuje obiekty operandów od lewej do prawej strony i zwraca pierwszy będący prawdą. Co więcej, po odnalezieniu pierwszego prawdziwego operandu Python zatrzymuje wyszukiwanie — takie zachowanie jest znane jako skrócona analiza typu *short-circuit* (czyli w przybliżeniu: po linii najmniejszego oporu — nawiązanie do tego, że ustalenie wyniku kończy analizę bez przetwarzania reszty wyrażenia).

```
>>> 2 or 3, 3 or 2 # Zwraca operand z lewej strony, jeśli jest
prawda,
(2, 3) # w przeciwnym wypadku zwraca operand z prawej
strony (prawda lub fałsz)

>>> [] or 3
3

>>> [] or {}
{}
```

W pierwszym wierszu powyższego przykładu oba obiekty (`2` oraz `3`) są prawdziwe (czyli są niezerowe), dlatego Python zatrzymuje się i zwraca obiekt z lewej strony — ustalenie wyniku jest od razu możliwe, ponieważ wynikiem wyrażenia logicznego `prawda OR cokolwiek` jest zawsze prawda. W dwóch kolejnych testach lewy operand jest fałszywy (pusty obiekt), dlatego Python analizuje i zwraca obiekt z prawej strony (który może być prawdziwy lub fałszywy).

Operacje `and` również zatrzymują się, kiedy tylko zostanie określony ich wynik. W tym przypadku Python sprawdza operandy od lewej do prawej strony i zatrzymuje się na pierwszym obiekcie będącym *fałszem*, ponieważ taki operand od razu determinuje rezultat wyrażenia — wynikiem wyrażenia `fałsz AND cokolwiek` jest zawsze `fałsz`.

```
>>> 2 and 3, 3 and 2 # Zwraca operand z lewej strony wyrażenia,
jeżeli jest fałszem,
(3, 2) # w przeciwnym wypadku zwraca operand z prawej
strony (prawda lub fałsz)

>>> [] and {}
[]

>>> 3 and []
[]
```

W powyższym przykładzie oba operandy z pierwszego wiersza są prawdą, dlatego Python analizuje obie strony i zwraca obiekt znajdujący się z prawej. W drugim teście lewy operand jest fałszem (`[]`), dlatego Python zatrzymuje się i zwraca go jako wynik testu. W ostatnim teście

lewa strona jest prawdą (3), dlatego Python analizuje i zwraca obiekt z prawej (którym okazuje się [] będące fałszem).

Rezultat jest taki sam jak w C i większości innych języków programowania — otrzymujemy wartość, która jest logiczną prawdą lub fałszem, kiedy przetestuje się ją w `if` lub `while` zgodnie z normalnymi definicjami operacji `and` i `or`. W Pythonie operacje logiczne zwracają jednak obiekty (lewy lub prawy), a nie po prostu liczbowe odpowiedniki prawdy lub fałszu.

Takie zachowanie operatorów `and` i `or` może się na pierwszy rzut oka wydawać nieco dziwne, jednak w ramce „Warto pamiętać — operatory logiczne” na końcu niniejszego rozdziału znajdziesz przykłady sytuacji, w których może ono być przydatne dla programistów Pythona. W kolejnym podrozdziale pokażemy również często spotykany sposób wykorzystania tego zachowania, a także jego bardziej mnemonicznego zamiennika w nowszych wersjach Pythona.

Wyrażenie trójargumentowe `if/else`

Jednym z często spotykanych zadań operatorów logicznych jest zapisywanie w kodzie programu wyrażeń działających tak samo jak instrukcje `if`. Rozważmy poniższą instrukcję, ustawiającą zmienną `A` na `Y` lub `Z` w zależności od tego, czy `X` jest prawdziwe:

```
if X:  
    A = Y  
else:  
    A = Z
```

Czasami jednak elementy takiej instrukcji są tak proste, że rozciąganie ich na cztery wiersze wydaje się być przesadą. Innym razem konieczne może się okazać zagnieżdżenie takiej konstrukcji w większej zamiast przypisywania jej wyniku do zmiennej. Z tego powodu (a poza tym dlatego, że podobne narzędzie znajduje się również w języku C) w Pythonie 2.5 wprowadzono nowy format instrukcji pozwalający na napisanie tego samego w jednym wyrażeniu:

```
A = Y if X else Z
```

Wyrażenie to ma dokładnie ten sam efekt jak poprzednia czterowierszowa instrukcja `if`, jednak jest łatwiejsze do zapisania. Tak jak w instrukcji `if`, Python wykonuje wyrażenie `Y` tylko wtedy, gdy `X` okazuje się prawdą, i wykonuje wyrażenie `Z` tylko wtedy, gdy `X` okazuje się fałszem. Jest to zatem skrót — charakterystyczny dla opisanych wcześniej operacji wykonywanych przez operatory Boolean. Poniżej widać przykłady zastosowania tego wyrażenia.

```
>>> A = 't' if 'mielonka' else 'f'          # Niepusty ciąg znaków ma  
wartość true, czyli prawda  
>>> A  
't'  
>>> A = 't' if '' else 'f'  
>>> A  
'f'
```

Przed wersją 2.5 (i nawet już po jej opublikowaniu) ten sam efekt można często osiągnąć po uważnym połączeniu operatorów `and` oraz `or`, ponieważ zwracają one albo obiekt z lewej strony, albo ten z prawej.

```
A = ((X and Y) or Z)
```

Taki kod działa, jednak jest w nim pewna pułapka — musimy założyć, że Y będzie prawdą. W tym przypadku rezultat będzie taki sam — najpierw działa operator `and` zwracający Y, jeśli X jest prawdą. Jeżeli tak nie jest, operator `or` zwraca po prostu Z. Innymi słowy, otrzymujemy „jeśli X, to Y; w przeciwnym razie Z”. Jest to odpowiednik trójargumentowego polecenia `if`:

```
A = Y if X else Z
```

Połączenie `and` i `or` wydaje się również wymagać chwili zastanowienia, by zrozumieć je za pierwszym razem, kiedy je zobaczymy. Od wersji 2.5 nie ma już takiej potrzeby — jeżeli potrzebne jest nam w postaci wyrażenia, wystarczy użyć równoważnego i łatwiejszego do zapamiętania `Y if X else Z`; można również użyć pełnej instrukcji `if/else`, jeżeli jej części są bardziej złożone.

Na marginesie można dodać, że podobny efekt da w Pythonie użycie poniższego wyrażenia, ponieważ funkcja `bool` tłumaczy X na odpowiednik liczb całkowitych 1 lub 0, który można następnie wykorzystać do wybrania wartości prawdy lub fałszu z listy.

```
A = [Z, Y][bool(X)]
```

Obrazuje to poniższy przykład.

```
>>> ['f', 't'][bool('')]  
'f'  
>>> ['f', 't'][bool('mielonka')]  
't'
```

Nie jest to jednak dokładnie to samo, ponieważ Python nie zastosuje w tym przypadku skrótu — zawsze wykonane zostaną zarówno Z, jak i Y, bez względu na wartość X. Ze względu na te komplikacje od wersji 2.5 lepiej jest użyć prostszego i łatwiejszego do zrozumienia wyrażenia `if/else`. Nawet z tego wyrażenia lepiej jest jednak korzystać oszczędnie i tylko wtedy, gdy jego części są stosunkowo proste. W innym przypadku zaleca się napisanie pełnej instrukcji `if`, co ułatwi wprowadzanie modyfikacji do kodu w przyszłości. Twórcy współpracownicy również ucieszą się z takiego wyboru.

W kodzie napisanym dla wersji Pythona starszych od 2.5 (a także pisanym przez byłych programistów języka C, którzy nie pozbawili się jeszcze swoich mrocznych nawyków programistycznych z przeszłości) nadal można jeszcze spotkać wersje z operatorami `and` oraz `or`[\[2\]](#).

Warto pamiętać — operatory logiczne

Jednym z często spotykanych zastosowań nieco nietypowego zachowania operatorów Boolean Pythona jest wybór ze zbioru obiektów za pomocą operatora `or`. Instrukcja podobna do poniższej:

```
X = A or B or C or None
```

ustawia X na pierwszy niepusty (będący prawdą) obiekt spośród A, B i C lub na `None`, jeśli wszystkie obiekty są puste. Powyższy kod działa, ponieważ operator `or` zwraca jeden z dwóch swoich obiektów. Technika ta okazuje się stosunkowo często stosowana w Pythonie. By wybrać niepusty obiekt ze zbioru o stałym rozmiarze, wystarczy połączyć obiekty w jednym wyrażeniu `or`. W prostszej postaci jest to często wykorzystywane do oznaczenia wartości domyślnej — poniższy kod ustawia X na A, jeśli A jest prawdziwe (lub niepuste), a w innym przypadku na `default`:

```
X = A or default
```

Zrozumienie skrótoowej analizy obiektów jest dość istotne, ponieważ wyrażenia znajdujące się po prawej stronie operatora Boolean mogą wywoływać funkcje wykonujące znaczące działania lub dawać efekty uboczne, które nie wystąpią, jeśli zastosowana zostanie analiza skrócona.

```
if f1() or f2(): ...
```

W powyższym kodzie, jeśli f1 zwraca wartość będącą prawdą (niepustą), Python nigdy nie zwróci f2. By zagwarantować, że wykonane zostaną obie funkcje, musimy je wywołać przed or.

```
tmp1, tmp2 = f1(), f2()
```

```
if tmp1 or tmp2: ...
```

W niniejszym rozdziale widzieliśmy już zastosowanie takiego zachowania. Ze względu na sposób działania operatorów Boolean wyrażenie ((A and B) or C) można wykorzystać do prawie dokładnego emulowania instrukcji if/else (bardziej szczegółowy opis działania tej postaci znajduje się w treści rozdziału).

Dodatkowe przypadki użycia operatorów Boolean widzieliśmy już w poprzednich rozdziałach. Zgodnie z informacjami z rozdziału 9., ponieważ każdy obiekt jest albo prawdą, albo fałszem, w Pythonie często łatwiej jest testować bezpośrednio obiekt (if X:), niż porównywać go z pustą wartością (if X != ' '). W przypadku łańcuchów znaków te dwa testy są równoważne. Jak wiemy także z rozdziału 5., odgórnie zdefiniowane wartości Boolean True oraz False są tym samym co liczby całkowite 1 oraz 0 i przydają się do inicjalizowania zmiennych (X = False), testów pętli (while True:), a także wyświetlania wyników w sesji interaktywnej.

Warto także zwrócić uwagę na omówienie przeciążania operatorów w szóstej części książki. Kiedy definiujemy nowe typy obiektów za pomocą klas, możemy określić ich naturę Boolean za pomocą metod __bool__ oraz __len__ (__bool__ w Pythonie 2.7 nosi nazwę __nonzero__). Druga z tych metod jest sprawdzana, jeśli nie ma pierwszej z nich, i przypisuje fałsz, zwracając wartość o długości zero — pusty obiekt uznawany jest za fałsz.

Wreszcie inne narzędzia w Pythonie mają role podobne do wyrażeń z operatorem OR pokazanych na początku tej ramki: wywołanie funkcji filter i listy składane, które spotkamy później, mogą zostać użyte do wybierania prawdziwych wartości, gdy zbiór kandydatów nie jest znany do czasu uruchomienia programu, a wbudowane operatory any oraz all mogą być używane do sprawdzania, czy którykolwiek lub wszystkie elementy w kolekcji są prawdziwe (chociaż nie wybierają żadnych elementów kolekcji):

```
>>> L = [1, 0, 2, 0, 'mielonka', '', 'szynka', []]
>>> list(filter(bool, L))                                     # Pobiera elementy o
wartości true
[1, 2, 'mielonka', 'szynka']
>>> [x for x in L if x]                                       # Składanie
[1, 2, 'mielonka', 'szynka']
>>> any(L), all(L)                                         # Agregacja prawdy
(True, False)
```

Jak pokazywaliśmy w rozdziale 9., funkcja bool zwraca po prostu prawdziwą lub fałszywą wartość argumentu, tak jakby była testowana za pomocą if.Więcej informacji na temat tych powiązanych narzędzi można znaleźć w rozdziałach 14., 19. i 20.

Podsumowanie rozdziału

W niniejszym rozdziale przedstawiliśmy instrukcję `if` Pythona. Co więcej, ponieważ była to pierwsza omawiana przez nas złożona instrukcja logiczna, przejrzieliśmy również ogólne reguły składni Pythona i pogłębiliśmy naszą wiedzę na temat wykonywania testów prawdziwości. Przy okazji przyjrzieliśmy się również sposobowi zapisu rozgałęzień kodu w Pythonie, dowiedzieliśmy się czegoś na temat wyrażenia `if/else` wprowadzonego w Pythonie 2.5 i omówiliśmy niektóre sposoby użycia operatorów logicznych.

W kolejnym rozdziale będziemy kontynuować omawianie instrukcji proceduralnych, poszerzając naszą wiedzę na temat pętli `for` oraz `while`. Nauczmy się alternatywnych sposobów tworzenia pętli w Pythonie; niektóre z nich będą lepsze od pozostałych. Przedtem jednak pora wykonać quiz podsumowujący niniejszy rozdział.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób można zapisać w Pythonie rozgałezienie kodu?
2. W jaki sposób można w Pythonie zapisać instrukcję `if/else` w postaci wyrażenia?
3. Jak można sprawić, by jedna instrukcja rozciągała się na kilka wierszy?
4. Co oznaczają słowa `True` i `False`?

Sprawdź swoją wiedzę — odpowiedzi

1. Instrukcja `if` z kilkoma częściami `elif` jest często najlepszym sposobem zapisania kodu rozgałęzionej, choć niekoniecznie jest przy tym najbardziej zwięzłą metodą. Podobny rezultat może często dać indeksowanie słowników, w szczególności kiedy słownik zawiera wywoływalne funkcje zapisane za pomocą instrukcji `def` czy wyrażeń `lambda`.
2. W Pythonie 2.5 i późniejszych wersjach wyrażenie w postaci `Y if X else Z` zwraca `Y`, jeżeli `X` jest prawdą; w przeciwnym razie zwraca `Z`. Jest ono odpowiednikiem czterowierszowej instrukcji `if`. Połączenie operatorów `and` oraz `or` — `((X and Y) or Z)` — może działać w ten sam sposób, jednak jest mniej oczywiste i wymaga, by część `Y` była prawdą.
3. Należy ująć instrukcję w parę nawiasów `((), []` lub `{ }`), dzięki czemu może ona rozciągać się na dowolną liczbę wierszy. Instrukcja kończy się, kiedy Python znajdzie zamkający, prawy nawias, a wiersze instrukcji od drugiego i dalej mogą się rozpoczynać na dowolnym poziomie wcięcia. Używanie znaków lewego ukośnika jako znaczników kontynuacji wierszy jest mocno odradzane we współczesnym świecie Pythona.
4. `True` i `False` są zapisanymi w innej postaci wersjami liczb całkowitych `1` i `0`. Zawsze oznaczają one w Pythonie wartości logiczne typu Boolean, reprezentujące odpowiednio prawdę i fałsz. Są one dostępne do użycia w testach prawdziwości oraz inicjalizacji zmiennych i wyświetlane jako wyniki wyrażeń w sesji interaktywnej. We

wszystkich wymienionych rolach służą one jako bardziej mnemoniczna, a zatem bardziej czytelna alternatywa dla wartości 1 i 0.

[1] Szczerze mówiąc, nieco zaskakujące było to, że kontynuacje wierszy przy użyciu znaków lewego ukośnika nie zostały usunięte w Pythonie 3.0, biorąc pod uwagę szeroki zakres innych zmian! Zobacz tabelę zmian wprowadzonych w wersji 3.0, którą znajdziesz w dodatku C; niektóre usunięte rozwiązania wydają się dość nieszkodliwe w porównaniu z niebezpieczeństwami związanymi z kontynuacją wiersza z użyciem lewego ukośnika. Niemniej celem tej książki jest przedstawienie Pythona, a nie populistyczne polemiki z deweloperami, więc najlepszą radą, jakiej mogę udzielić, jest po prostu: nie rób tego. Ogólnie powinieneś unikać kontynuacji wierszy z użyciem znaku lewego ukośnika w nowym kodzie Pythona, nawet jeżeli wyrobiliś sobie taki nawyk, programując w języku C.

[2] Tak naprawdę wyrażenie `X if Y else Z` ma w Pythonie nieco inną kolejność wykonywania niż `Y ? X : Z` z języka C i używa bardziej czytelnego zapisu. Podobno zmianę tę wprowadzono w oparciu o analizę wzorców użycia w kodzie Pythona. Wieść niesie, że kolejność ta została wybrana po części również dlatego, by zniechęcić byłych programistów języka C do nadużywania tego wzorca. Należy pamiętać, że proste jest lepsze od skomplikowanego, zarówno w Pythonie, jak i w każdej innej dziedzinie.

Rozdział 13. Pętle while i for

Niniejszy rozdział kończy nasze omówienie instrukcji proceduralnych Pythona, przedstawiając dwie najważniejsze konstrukcje *pętli* Pythona — instrukcji powtarzających jakieś działanie. Pierwsza z nich, instrukcja `while`, umożliwia zapisywanie w kodzie uniwersalnych pętli. Druga, instrukcja `for`, pozwala przechodzić przez kolejne elementy sekwencji lub innego obiektu iterowalnego i wykonywać blok kodu dla każdego z tych elementów.

Z obydwojoma instrukcjami spotkaliśmy się już w sposób nieformalny wcześniej, tutaj natomiast uzupełnimy informacje na temat ich stosowania. A skoro już przy tym jesteśmy, omówimy także kilka mniejszych popularnych instrukcji stosowanych wewnętrz pętli, takich jak `break` oraz `continue`, i przedstawimy kilka funkcji wbudowanych wykorzystywanych najczęściej w połączeniu z pętlami — jak `range`, `zip` czy `map`.

W Pythonie istnieją również inne operacje pętli, jednak te dwie instrukcje są podstawową składnią służącą do tworzenia kodu dla powtarzanych działań. Z tego powodu w kolejnym rozdziale zapoznamy się z powiązaną koncepcją *protokołu iteracji* Pythona (wykorzystywanego przez pętlę `for`) i uzupełnimy pewne szczegóły dotyczące *list składanych* — bliskiego kuzyna tej pętli. W dalszych rozdziałach będziemy omawiać jeszcze bardziej egzotyczne narzędzia iteracyjne, takie jak *generatory*, *filter* oraz *reduce*. Na razie jednak pozostaniemy przy nieco prostszych zagadnieniach.

Pętle while

Instrukcja `while` Pythona jest najbardziej uniwersalną konstrukcją iteracyjną tego języka. W uproszczeniu powtarza ona wykonywanie bloku (normalnie wciętych) instrukcji, dopóki test znajdujący się na górze zwraca wartość będącą prawdą. Nazywana jest „*pętlą*”, ponieważ sterowanie powraca ciągle do początku tej instrukcji, dopóki test nie zwróci fałszu. Kiedy tak się stanie, sterowanie przechodzi do instrukcji następującej po bloku `while`. W rezultacie ciało pętli wykonywane jest raz za razem, dopóki test znajdujący się w jej nagłówku zwraca prawdę. Jeśli test od początku będzie zwracał fałsz, ciało pętli nigdy nie zostanie wykonane.

Ogólny format

W najbardziej złożonej postaci instrukcja `while` składa się z wiersza nagłówka z wyrażeniem testowym, ciała zawierającego jedną lub większą liczbę wciętych instrukcji oraz opcjonalną część `else` wykonywaną, kiedy sterowanie opuszcza pętlę bez napotkania instrukcji `break`. Python powtarza analizowanie testu z nagłówka i wykonywanie instrukcji zagnieżdżonych w ciele pętli, dopóki test nie zwróci fałszu.

```
while test:                                # Warunek pętli
    instrukcje1                            # Ciało pętli
else:                                         # Opcjonalne else
    instrukcje2                            # Wykonane, jeśli pętli nie
    zakończyło break
```

Przykłady

By zilustrować działanie pętli `while`, przyjrzyjmy się kilku przykładom jej zastosowania. Pierwszy, składający się z instrukcji `print` zagnieździonej w pętli `while`, po prostu w nieskończoność wyświetla komunikat. Warto przypomnieć, że `True` to inaczej zapisana wersja liczby całkowitej 1 i zawsze oznacza wartość Boolean dla prawdy. Ponieważ test zawsze będzie zwracał prawdę, Python będzie wykonywał ciało pętli w nieskończoność — lub dopóki mu nie przerwiemy. Takie zachowanie nazywa się często *nieskończoną pętlą* — oczywiście taka pętla nie jest tak naprawdę nieskończona, ale możesz potrzebować kombinacji klawiszy `Ctrl+C`, aby wymusić jej zakończenie:

```
>>> while True:  
...     print('Naciśnij kombinację klawiszy Ctrl+C, by mnie zatrzymać!')
```

Kolejny przykład odcina ostatni znak łańcucha, dopóki łańcuch ten nie stanie się pusty i tym samym nie będzie fałszem. Zazwyczaj obiekty testuje się w sposób bezpośredni, zamiast korzystać z bardziej rozwlekłego odpowiednika `while x != ''`. W dalszej części rozdziału zobaczymy inne sposoby bezpośredniego przechodzenia elementów w łańcuchu znaków za pomocą pętli `for`.

```
>>> x = 'mielonka'  
>>> while x:  
...     print(x, end=' ')  
print x  
...     x = x[1:]  
...  
mielonka ielonka elonka lonka onka nka ka a
```

Warto tutaj zwrócić uwagę na argument ze słowem kluczowym `end=' '`, który pozwala na wyświetlenie wszystkich danych wyjściowych w jednym wierszu. Wracając do rozdziału 11., można dowiedzieć się, dlaczego tak się dzieje. Po wykonaniu tego programu znak zachęty może pozostać w nieco dziwnym stanie; jeżeli tak się stanie, naciśnij klawisz *Enter*, aby go zresetować. Jeżeli jesteś użytkownikiem Pythona 2.x, powinieneś również pamiętać, aby zamiast klauzuli `end` używać w takich wyrażeniach przecinka.

Kod przedstawiony w kolejnym przykładzie poniżej odlicza wartości od `a` do `b` (bez samego `b`). Później zobaczymy, że taki sam rezultat można w łatwiejszy sposób uzyskać za pomocą pętli `for` i wbudowanej funkcji `range`.

```
>>> a=0; b=10  
>>> while a < b:  
...     print(a, end=' ')  
...     a += 1  
...  
0 1 2 3 4 5 6 7 8 9
```

Wreszcie warto zwrócić uwagę na to, że w Pythonie nie ma instrukcji pętli typu `do until` (wykonuj, dopóki...). Możemy jednak taką instrukcję symulować za pomocą testu i instrukcji `break` na końcu ciała pętli, dzięki czemu ciało pętli zostanie zawsze wykonane co najmniej raz:

```
while True:
```

...ciało pętli...

```
if exitTest(): break
```

Aby w pełni zrozumieć działanie tej struktury, musimy przejść do kolejnego podrozdziału, w którym dowiemy się więcej o instrukcjach `break`.

Instrukcje `break`, `continue`, `pass` oraz `else` w pętli

Skoro już zobaczyliśmy kilka przykładów działających pętli Pythona, czas przyjrzeć się dwóm prostym instrukcjom, które mają sens dopiero wtedy, gdy zagnieżdżone są w pętlach — mowa tu o instrukcjach `break` oraz `continue`. A skoro już mowa o osobliwościach, omówimy przy okazji również blok `else` z pętli, ponieważ przeplata się on z `break`. Zajmiemy się także pustym pojemnikiem instrukcji, czyli `pass` (który nie jest powiązany z pętlami *per se*, ale mieści się w ogólnej kategorii prostych instrukcji składających się z jednego słowa). W Pythonie:

```
break
```

Wychodzi z najbliższej obejmującej daną instrukcję pętli (omija całą instrukcję pętli).

```
continue
```

Przechodzi na góre najbliższej obejmującej daną instrukcję pętli (do jej wiersza nagłówka).

```
pass
```

Nic nie robi. Jest pustym elementem zastępczym instrukcji.

Blok pętli else

Wykonywany jest wtedy (i tylko wtedy), gdy pętla kończy się normalnie — bez trafienia na instrukcję `break`.

Ogólny format pętli

Uwzględniając instrukcje `break` i `continue`, ogólny format pętli `while` będzie wyglądał następująco.

```
while test1:  
    instrukcje1  
    if test2: break                      # Wyjście z pętli, pominięcie  
    reszty  
    if test3: continue                   # Przejście do góry pętli, do  
    test1  
    else:  
        instrukcje2                      # Wykonywane, jeśli nie trafiłyśmy  
        na break
```

Instrukcje `break` i `continue` mogą się pojawiać w dowolnym miejscu ciała pętli `while` (lub `for`), jednak zazwyczaj umieszczane są jako zagnieżdżone w teście `if` i wykonują jakieś działanie w odpowiedzi na określony warunek.

W celu przekonania się, jak w praktyce łączą się ze sobą te instrukcje, warto przyjrzeć się kilku prostym przykładom.

Instrukcja pass

Na początek najprostsze. Instrukcja `pass` jest elementem zastępczym bez działania, wykorzystywany wtedy, gdy składnia wymaga instrukcji, ale nie mamy nic użytecznego do powiedzenia. Często wykorzystywana jest do zapisywania w kodzie pustego ciała instrukcji złożonej. Jeśli na przykład chcemy umieścić w kodzie nieskończoną pętlę, która przy każdym przejściu nic nie robi, możemy skorzystać z instrukcji `pass`.

```
while True: pass # Użyj Ctrl+C, by mnie zatrzymać!
```

Ponieważ ciało pętli jest pustą instrukcją, Python zacina się w tej pętli. Instrukcja `pass` jest w instrukcjach mniej więcej tym, czym `None` dla obiektów — jawnie wyrażonym niczym. Warto zauważyc, że w powyższym przykładzie ciało pętli `while` umieszczone jest w jednym wierszu z jej nagłówkiem, po dwukropku. Tak jak w przypadku instrukcji `if`, taki zapis działa wyłącznie wtedy, gdy ciało nie jest instrukcją złożoną.

Powyższy przykład powoduje, że Python wchodzi w nieskończoną pętlę, w której nie wykonuje żadnego działania. Nie jest to zatem najbardziej użyteczny program w historii programowania w Pythonie (o ile nie wykorzystamy go do rozgrzania naszego laptopa w chłodny, zimowy dzień). Szczerze mówiąc, po prostu nie umiałem wymyślić lepszego przykładu z `pass` na tym etapie książki.

Później zobaczymy inne sytuacje, w których `pass` ma większy sens — na przykład w ignorowaniu wyjątków przechyconych za pomocą instrukcji `try`, definiowaniu pustych obiektów `class` z atrybutami zachowującymi się jak struktury i rekordy z innych języków programowania. Instrukcja `pass` czasami oznacza w kodzie miejsce, które zostanie uzupełnione później, a także służy do tymczasowego wyróżnienia ciała funkcji.

```
def func1():
    pass # Wstawić tu później prawdziwy kod!
def func2():
    pass
```

Nie możemy pozostawić ciała funkcji pustego bez otrzymania błędu składni, dlatego zamiast tego wstawiamy do niego instrukcję `pass`.



Uwaga na temat wersji: Python 3.x (ale wersja 2.x już nie) pozwala na wpisywanie w kodzie `elips` w postaci `...` (dosłownie: trzech następujących po sobie kropek) we wszystkich miejscach, w których może się pojawić wyrażenie. Ponieważ elipsy same z siebie nic nie robią, mogą służyć jako alternatywa dla instrukcji `pass`, zwłaszcza w przypadku kodu, który chcemy uzupełnić później:

```
def func1():
    ...
# Alternatywa dla pass
def func2():
    ...
func1() # Po wywołaniu nic nie robi
```

Elipsy mogą się także pojawiać w tym samym wierszu co nagłówek instrukcji i można je wykorzystywać do inicjalizacji nazw zmiennych, jeśli nie jest wymagany ich określony typ:

```
def func1(): ... # Działa także w tym
                  samym wierszu

def func2(): ...

>>> X = ... # Alternatywa dla None

>>> X

Ellipsis
```

Powyższy zapis jest nowością w Pythonie 3.x (i wychodzi znacznie poza początkowe zastosowanie znaków `...` w rozszerzeniach związanych z wycinkami). Z czasem okaże się, czy stanie się na tyle popularny, by zagrozić pozycji `pass` oraz `None` w tych zastosowaniach.

Instrukcja `continue`

Instrukcja `continue` powoduje natychmiastowe przejście na góre pętli. Czasami pozwala nam uniknąć zagnieźdzania instrukcji. Poniższy przykład wykorzystuje instrukcję `continue` do pomijania nieparzystych liczb. Kod ten wyświetla wszystkie parzyste liczby mniejsze od 10 i większe od lub równe 0. Należy pamiętać, że 0 oznacza fałsz, a `%` jest operatorem reszty z dzielenia, dlatego pętla ta odlicza od 10 do 0, pomijając liczby niebędące wielokrotnościami 2 (wyświetla zatem 8 6 4 2 0).

```
x = 10

while x:
    x = x-1 # Lub x -= 1
    if x % 2 != 0: continue # Nieparzyste? Pomijamy!
    print(x, end=' ')
```

Ponieważ `continue` powoduje przeskoczenie na góre pętli, nie musimy zagnieźdzać instrukcji `print` wewnętrz testu `if`. Do instrukcji `print` dochodzimy dopiero wtedy, gdy nie zostanie wykonana instrukcja `continue`. Jeśli brzmi to podobnie do formy `goto` z innych języków programowania, to dobrze — tak właśnie powinno być. W Pythonie nie ma instrukcji `goto`, jednak ponieważ `continue` pozwala na przeskoki w programie, wiele z ostrzeżeń dotyczących czytelności i możliwości utrzymywania kodu dotyczących `goto` ma zastosowanie również tutaj. Instrukcji `continue` powinno się używać oszczędnie, w szczególności na początku naszej przygody z Pythonem. Przykład wyżej mógłby być nieco jaśniejszy, gdyby instrukcja `print` została zagnieźdzona pod `if`.

```
x = 10

while x:
    x = x-1
    if x % 2 == 0: # Parzyste? Wyświetlamy!
        print(x, end=' ')
```

W dalszej części tej książki dowiemy się również, że zgłasiane i przechwycone wyjątki mogą naśladować wyrażenia `goto` w ograniczony i uporządkowany sposób; więcej szczegółowych

informacji na ten temat znajdziesz w rozdziale 36., gdzie nauczysz się, jak ich używać do wychodzenia z wielu zagnieźdzonych pętli, co nie jest możliwe przy użyciu samej instrukcji `break`, omawianej w następnej sekcji.

Instrukcja `break`

Instrukcja `break` powoduje natychmiastowe wyjście z pętli. Ponieważ kod następujący po niej w pętli nie zostanie wykonany, dzięki umieszczeniu w kodzie instrukcji `break` czasami można uniknąć zagnieźdzania. Poniżej znajduje się prosta pętla interaktywna (wariant większego przykładu omawianego w rozdziale 10.), której dane wejściowe uzyskuje się za pomocą funkcji `input` (w Pythonie 2.x znanej pod nazwą `raw_input`) i która kończy się, kiedy użytkownik w odpowiedzi na żądanie podania imienia wpisze „stop”.

```
>>> while True:  
...     name = input('Podaj imię:')                      # W wersji 2.x użyj  
funkcji raw_input()  
...     if name == 'stop': break  
...     age = input('Podaj wiek: ')  
...     print('Witaj,', name, '=>', int(age) ** 2)  
  
...  
Podaj imię:Amadeusz  
Podaj wiek: 40  
Witaj, Amadeusz => 1600  
Podaj imię:Maurycy  
Podaj wiek: 30  
Witaj, Maurycy => 900  
Podaj imię:stop
```

Warto zwrócić uwagę na sposób konwersji danych wejściowych z wiekiem na liczbę całkowitą przed podniesieniem ich do kwadratu. Jak pamiętamy, jest to konieczne, ponieważ funkcja `input` zwraca dane od użytkownika w postaciłańca znaków. W rozdziale 36. zobaczymy, że `input` zwraca również wyjątek na końcu pliku (na przykład kiedy użytkownik skorzysta ze skrótu *Ctrl+Z* w systemie Windows lub *Ctrl+D* w systemie Unix i podobnych). Jeżeli będzie to mogło mieć wpływ na nasz kod, należy umieścić tę funkcję w instrukcjach `try`.

Klauzula `else` pętli

W połączeniu z klauzulą `else` pętli instrukcja `break` może często wyeliminować konieczność używania opcji (flag) statusu wyszukiwania z innych języków programowania. Poniższy fragment kodu ustala na przykład, czy dodatnia liczba całkowita `y` jest liczbą pierwszą, wyszukując czynniki większe od 1.

```
x = y // 2                                              # Dla jakiegoś y > 1  
while x > 1:  
    if y % x == 0:                                       # Reszta  
        print(y, 'ma czynnik', x)
```

```

break                                # Pominiecie reszty
x -= 1
else:                                 # Normalne wyjście
    print(y, 'jest liczbą pierwszą')

```

Zamiast ustawać opcję statusu, która ma być sprawdzana przy wyjściu z pętli, wystarczy wstawić instrukcję `break` w miejsce, gdzie odnaleziony zostaje czynnik. W ten sposób klauzula `else` zakłada, że będzie wykonana tylko wtedy, jeśli żaden czynnik nie zostanie odnaleziony. Jeżeli nie trafimy na `break`, liczba jest liczbą pierwszą. Spróbuj uważnie przeanalizować kod przykładowy, aby zrozumieć, jak on działa.

Klauzula `else` wykonywana jest również wtedy, gdy ciało pętli nigdy nie zostanie wykonane, gdyż w takim przypadku również nie wykonamy instrukcji `break`. W pętli `while` może się tak stać, jeżeli test z nagłówka od początku będzie fałszem. W rezultacie w przykładzie tym otrzymamy komunikat „jest liczbą pierwszą”, jeżeli `x` początkowo będzie mniejsze lub równe 1 (na przykład kiedy `y` ma wartość 2).



Przykład oblicza liczby pierwsze, ale tylko w sposób nieformalny. Liczby mniejsze od 2 nie są w ścisłej definicji matematycznej uważane za liczby pierwsze. Zwróć uwagę, że ten kod nie będzie również działał dla liczb ujemnych, za to zadziała dla liczb zmiennoprzecinkowych bez miejsc dziesiętnych. Warto również zauważać, że kod musi w Pythonie 3.x wykorzystywać operator `//` zamiast `/` z uwagi na przejście z dzielenia `/` na „prawdziwe dzielenie” opisane w rozdziale 5. (początkowe dzielenie musi odciąć resztę, a nie ją zachować!). Jeżeli chcesz samodzielnie eksperymentować z kodem przykładowym, powinieneś zajrzeć do ćwiczenia na końcu czwartej części książki, w którym opakujemy ten kod w funkcję.

Więcej o części pętli `else`

Ponieważ klauzula `else` pętli jest unikatowa dla Pythona, może wprawiać w zakłopotanie wielu początkujących użytkowników tego języka (i często pozostaje niewykorzystana przez wielu weteranów; spotkałem takich, którzy nawet nie wiedzieli, że w pętlach może występować klauzula `else!`). Ogólnie mówiąc, klauzula `else` pętli zapewnia po prostu jawną składnię dla typowego scenariusza programowania — jest to struktura, która pozwala nam na „inne” wyjście z pętli, bez ustawiania i sprawdzania flag lub warunków.

Załóżmy na przykład, że tworzymy pętlę przeszukującą listę pod kątem wartości i po zakończeniu pętli musimy wiedzieć, czy dana wartość została odnaleziona. Takie zadanie można zapisać w kodzie w sposób pokazany poniżej (przedstawiony kod jest celowo abstrakcyjny i niekompletny; `x` to sekwencja, a `match` to funkcja testująca, którą należy zdefiniować):

```

found = False
while x and not found:
    if match(x[0]):                      # Wartość na początku?
        print('Ni')
        found = True
    else:
        x = x[1:]                         # Odcięcie początku i
        powtórzenie
if not found:

```

```
print('Nie znaleziono')
```

W powyższym kodzie inicjalizujemy, ustawiamy i później sprawdzamy opcję statusu w celu określenia, czy wyszukiwanie zakończyło się powodzeniem. Jest to poprawny kod Pythona, który działa. Jest to również rodzaj struktury, w której idealnie przydaje się klauzula `else`. Poniżej widać jej odpowiednik.

```
while x:                                # Wyjście, gdy x jest puste
    if match(x[0]):
        print('Ni')
        break                               # Wyjście, ominięcie else
    x = x[1:]
else:
    print('Nie znaleziono')              # Tylko wtedy, gdy wyczerpano x
```

Ta wersja jest bardziej zwięzła. Opcja statusu znika, a test `if` na końcu pętli zastąpiono za pomocą klauzuli `else` (wyrównanej w pionie ze słowem `while`). Ponieważ instrukcja `break` w środku głównej części `while` powoduje wyjście z pętli i obejście `else`, rozwiązanie to jest bardziej ustrukturyzowanym sposobem przechwycenia przypadku niepowodzenia wyszukiwania.

Niektóre osoby mogły zauważyc, że w poprzednim przykładzie klauzulę `else` można było zastąpić sprawdzeniem pustego `x` po pętli (na przykład `if not x:`). Choć w tym przykładzie jest to prawda, klauzula `else` udostępnia jawną składnię dla tego rodzaju wzorca kodu (ewidentnie mamy tutaj do czynienia z kodem przechwytyującym niepowodzenie wyszukiwania), a jawnie sprawdzenie pustego obiektu może w niektórych przypadkach nie mieć zastosowania. Klauzula `else` staje się nawet bardziej przydatna w połączeniu z pętlą `for`, która będzie tematem kolejnego podrozdziału, ponieważ iteracja po sekwencji pozostaje poza naszą kontrolą.

Warto pamiętać — emulacja pętli while z języka C

W części poświęconej instrukcjom wyrażeń w rozdziale 11. pisaliśmy, że Python nie pozwala instrukcjom takim, jak przypisanie, na pojawianie się w miejscach, w których oczekiwane jest wyrażenie. Oznacza to, że każda instrukcja musi generalnie występować sama w wierszu, a nie być zagnieżdżona w większej konstrukcji, co implikuje, że ten popularny wzorzec kodowania w języku C nie działa w Pythonie:

```
while ((x = next(obj)) != NULL) {...przetwarzanie x...}
```

Przypisania z języka C zwracają przypisaną wartość, natomiast przypisania w Pythonie są po prostu instrukcjami, a nie wyrażeniami. Eliminuje to pewną klasę notorycznie popełnianych błędów z języka C (nie można w Pythonie przypadkowo wpisać `=`, kiedy tak naprawdę mamy na myśli `==`). Jeżeli jednak zależy nam na podobnym zachowaniu, istnieją co najmniej trzy sposoby otrzymania tego samego efektu w Pythonie za pomocą pętli `while`, bez osadzania przypisania w testach pętli. Można przenieść przypisanie do ciała pętli z wykorzystaniem instrukcji `break`:

```
while True:
    x = next(obj)
    if not x: break
    ...przetwarzanie x...
lub przesunąć przypisanie do pętli z testami:
x = True
```

```
while x:  
    x = next(obj)  
    if x:  
        ...przetwarzanie x...  
albo przesunąć pierwsze przypisanie poza pętlę:  
  
x = next(obj)  
while x:  
    ...przetwarzanie x...  
    x = next(obj)
```

Z tych trzech wzorców kodu pierwszy może być przez niektórych uznawany za najmniej ustrukturyzowany, jednak wydaje się jednocześnie najprostszy i w praktyce jest najczęściej używany. Prosta pętla `for` może również zastąpić takie pętle z języka C, choć w języku C nie ma bezpośrednio analogicznego narzędzia:

```
for x in obj: ...przetwarzanie x...\n
```

Pętle `for`

Pętla `for` jest w języku Python uniwersalnym iteratorem — może przechodzić przez kolejne elementy w dowolnej uporządkowanej sekwencji lub innym obiekcie iterowalnym. Instrukcja `for` działa na łańcuchach znaków, listach, krotkach, innych wbudowanych obiektach, po których można iterować, i nowych obiektach definiowanych przez użytkownika, które — jak zobaczymy później — można tworzyć za pomocą klas. Z pętlami `for` spotkaliśmy się krótko w rozdziale 4. oraz podczas omawiania typów obiektów sekwencji, zatem nadszedł czas, aby rozwinąć teraz bardziej formalnie ich sposób użycia.

Ogólny format

Pętla `for` w Pythonie rozpoczyna się od wiersza nagłówka określającego cel (lub cele) przypisania wraz z obiektem, który chcemy przechodzić. Po nagłówku znajduje się blok (normalnie wciętych) instrukcji, które chcemy powtórzyć.

```
for cel in obiekt:                      # Przypisanie elementów obiektu do  
celu  
  
    instrukcje                         # Powtarzane ciało pętli: użycie  
celu  
  
else:                                     # Opcjonalna klauzula else  
  
    instrukcje                         # Jeśli nie trafiliśmy na break
```

Kiedy Python wykonuje pętlę `for`, jeden po drugim przypisuje elementy z obiektu iterowalnego do celu i wykonuje dla każdego z nich ciało pętli. Ciało pętli zazwyczaj wykorzystuje cel przypisania do odniesienia się do bieżącego elementu sekwencji, tak jakby był on kursorem przechodzącym sekwencję.

Nazwa użyta jako cel przypisania w wierszu nagłówka jest zazwyczaj (nową) zmienną w zakresie, w którym tworzona jest instrukcja `for`. Nie ma w niej nic specjalnego — może ona nawet zostać zmieniona w ciele pętli, jednak automatycznie zostanie ustawiona na kolejny element sekwencji, kiedy sterowanie powróci znowu na góre pętli. Po pętli zmienna najczęściej nadal odnosi się do ostatniego przechodzonego elementu (i tym samym ostatniego elementu sekwencji), o ile pętla nie zakończyła się na instrukcji `break`.

Instrukcja `for` obsługuje również opcjonalny blok `else`, który działa dokładnie tak samo jak w pętli `while` — jest wykonywany wtedy, gdy pętla kończy się bez trafienia na instrukcję `break` (to znaczy przetworzone zostały wszystkie elementy sekwencji). Omówione wcześniej instrukcje `break` i `continue` również w pętlach `for` działają w ten sam sposób co w `while`. Pełny format pętli `for` może zatem zostać przedstawiony w następujący sposób.

```
for cel in obiekt:                                # Przypisanie elementów obiektu do
    celu

    instrukcje

    if test: break                                  # Wyjście z pętli, pominięcie else
    if test: continue                               # Przejście na góre pętli

else:
    instrukcje                                     # Jeśli nie trafiiliśmy na break
```

Przykłady

Aby zobaczyć, jak pętle `for` zachowują się w praktyce, wpiszmy teraz kilka przykładów do sesji interaktywnej.

Podstawowe zastosowanie

Jak wspomniano wcześniej, pętla `for` może przechodzić przez dowolny obiekt sekwencji. W naszym pierwszym przykładzie przypiszemy na przykład zmienną `x` do każdego z trzech elementów listy po kolei, od lewej do prawej strony. Dla każdego z nich zostanie wykonana instrukcja `print`. Wewnątrz instrukcji `print` (w ciele pętli) zmienna `x` odnosi się do bieżącego elementu listy.

```
>>> for x in ["mielonka", "jajka", "szynka"]:
...     print(x, end=' ')
...
mielonka jajka szynka
```

Kolejne dwa przykłady obliczają sumę i iloczyn wszystkich elementów listy. W dalszej części tego rozdziału i książki spotkamy się z narzędziami automatycznie stosującymi operacje, takie jak `+` oraz `*`, do elementów listy. Zazwyczaj jest to tak samo proste jak użycie pętli `for`.

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
```

```
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24
```

Inne typy danych

W połączeniu z pętlą `for` można użyć dowolnego typu sekwencji, ponieważ narzędzie to jest uniwersalne. Pętle `for` działają na przykład na łańcuchach znaków i krotkach.

```
>>> S = "drwal"
>>> T = ("i", "jestem", "git")
>>> for x in S: print(x, end=' ')
# Iteracja po łańcuchu znaków
...
d r w a l
>>> for x in T: print(x, end=' ')
# Iteracja po krotce
...
i jestem git
```

Tak naprawdę, jak zobaczymy w kolejnym rozdziale przy okazji omawiania obiektów, na których można wykonywać iterację, pętle `for` działają nawet na pewnych obiektach, które nie są sekwencjami — w tym na plikach i słownikach!

Przypisanie krotek w pętli for

Jeśli wykonujemy iterację na sekwencji krotek, sam cel pętli może być *krotką* celów. To kolejny przykład działania mechanizmu przypisania rozpakowującego krotki, omówionego w rozdziale 11. Należy pamiętać, że pętla `for` przypisuje elementy obiektu sekwencji do celu, a przypisanie wszędzie działa tak samo.

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:
# Przypisanie krotek
...     print(a, b)
...
1 2
3 4
5 6
```

W powyższym kodzie pierwsze przejście pętli odpowiada zapisowi $(a, b) = (1, 2)$, drugie — $(a, b) = (3, 4)$ i tak dalej. W rezultacie z każdą iteracją automatycznie wypakowujemy bieżącą krotkę.

Ta forma wykorzystywana jest często w połączeniu z wywołaniem `zip`, z którym spotkamy się w dalszej części niniejszego rozdziału przy okazji implementacji przechodzenia równoległego. Regularnie pojawia się także w Pythonie w połączeniu z bazami danych SQL, w których tablice wyników zapytań zwracane są w postaci sekwencji sekwencji, podobnych do wykorzystanej

tutaj listy. Zewnętrzna lista to tabela bazy danych, zagnieżdżone krotki to wiersze wewnątrz tej tabeli, a przypisanie krotek powoduje ekstrakcję kolumn.

Krotki w pętlach `for` przydają się także do iteracji po kluczach i wartościach słowników za pomocą metody `items` zastępującej pętlę po kluczach i indeksowanie w celu ręcznego pobrania wartości:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
...     print(key, '=>', D[key])                      # Użycie iteratora po kluczach
słownika i indeksowania
...
a => 1
c => 3
b => 2
>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]
>>> for (key, value) in D.items():
...     print(key, '=>', value)                      # Iteracja po kluczach i
wartościach
...
a => 1
c => 3
b => 2
```

Należy koniecznie zauważyć, że przypisanie krotek w pętlach `for` nie jest żadnym przypadkiem specjalnym. Każdy cel przypisania z punktu widzenia składni zadziała po słowie `for`. Zawsze możemy jednak wykonać przypisanie ręcznie wewnątrz pętli w celu rozpakowania:

```
>>> T
[(1, 2), (3, 4), (5, 6)]
>>> for both in T:
...     a, b = both                                # Odpowiednik z przypisaniem
ręcznym
...     print(a, b)                                # w wersji 2.x: wyświetla
wartości z krotkami ujętymi w nawiasy
...
1 2
3 4
5 6
```

Użycie krotki w nagłówku pętli oszczędza nam dodatkowego kroku przy iteracji po sekwencjach sekwencji. Zgodnie z informacjami z rozdziału 11. w instrukcji `for` można w ten sposób automatycznie rozpakowywać nawet struktury *zagnieżdżone*:

```

>>> ((a, b), c) = ((1, 2), 3)          # Sekwencje zagnieżdżone też
działaają

>>> a, b, c

(1, 2, 3)

>>> for ((a, b), c) in [(1, 2), 3], ((4, 5), 6)]: print(a, b, c)

...
1 2 3

4 5 6

```

Nie jest to jednak żaden przypadek specjalny — pętla `for` po prostu wykonuje przypisanie, które wykonaliśmy tuż przed nią, z każdą iteracją. W ten sposób można rozpakować dowolną strukturę sekwencji, właśnie dlatego, że *przypisanie sekwencji* jest tak uniwersalne.

```

>>> for ((a, b), c) in [[1, 2], 3], ['XY', 6]: print(a, b, c)

...
1 2 3

X Y 6

```

Rozszerzone przypisanie sekwencji w pętlach for w Pythonie 3.x

Tak naprawdę, ponieważ zmienna pętli w pętli `for` może być dowolnym celem przypisania, możemy tutaj wykorzystać także składnię rozszerzonego przypisania rozpakowującego sekwencję z Pythona 3.x w celu dokonania ekstrakcji elementów i części sekwencji wewnętrz sekwencji. W rzeczywistości to także nie jest żaden przypadek specjalny, a po prostu nowa forma przypisania z wersji 3.x (omówiona w rozdziale 11.). Ponieważ działa ona w instrukcjach przypisania, automatycznie działa także w pętlach `for`.

Rozważmy raz jeszcze kod z przypisaniem krotki przedstawiony wyżej. Krotka wartości przypisywana jest do krotki nazw z każdą iteracją, dokładnie tak samo jak w prostej instrukcji przypisania:

```

>>> a, b, c = (1, 2, 3)                  # Przypisanie krotki

>>> a, b, c

(1, 2, 3)

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:      # Wykorzystane w pętli
for
...     print(a, b, c)

...
1 2 3

4 5 6

```

Ponieważ sekwencję można przypisać do bardziej uniwersalnego zbioru nazw za pomocą nazwy ze znakiem `*` służącej do zbierania większej liczby elementów, w Pythonie 3.x możemy wykorzystać tę samą składnię do ekstrakcji części zagnieżdżonych sekwencji w pętli `for`:

```

>>> a, *b, c = (1, 2, 3, 4)              # Rozszerzone
przypisanie sekwencji

>>> a, b, c

```

```
(1, [2, 3], 4)
>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
...     print(a, b, c)
...
1 [2, 3] 4
5 [6, 7] 8
```

W praktyce rozwiązywanie to można wykorzystać do wybrania kilku kolumn z wierszy danych reprezentowanych jako zagnieżdżone sekwencje. W Pythonie 2.X nazwy ze znakiem * nie są dozwolone, jednak podobny efekt można uzyskać za pomocą wycinków. Jedyna różnica polega na tym, że wycinek zwraca wynik specyficzny dla typu, natomiast do nazw ze znakiem * zawsze przypisywane są listy.

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:           # Ręczne wycinki z
Pythona 2.x
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
...
1 (2, 3) 4
5 (6, 7) 8
```

Więcej informacji na temat tej formy przypisania można znaleźć w rozdziale 11.

Zagnieżdżone pętle for

Przyjrzyjmy się teraz pętli for nieco bardziej wyszukanej od zaprezentowanych dotychczas. Kolejny przykład ilustruje zastosowanie w pętli for części else, a także zagnieżdżanie instrukcji. Mając podaną listę obiektów (items) i listę kluczy (tests), kod ten wyszukuje w liście obiektów każdego klucza i zgłasza rezultat tego działania.

```
>>> items = ["aaa", 111, (4, 5), 2.01]      # Zbiór obiektów
>>> tests = [(4, 5), 3.14]                  # Szukane klucze
>>>
>>> for key in tests:                      # Dla wszystkich kluczy
...     for item in items:                   # Dla wszystkich elementów
...         if item == key:                  # Sprawdzenie dopasowania
...             print(key, "znaleziono")
...             break
...     else:
...         print(key, "nie znaleziono!")
...
(4, 5) znaleziono
3.14 nie znaleziono!
```

Ponieważ zagnieźdzona instrukcja `if` wykonuje instrukcję `break`, kiedy klucz zostaje dopasowany, część `else` zakłada, że jeśli się do niej dojdzie, to wyszukiwanie nie powiodło się. Warto zwrócić uwagę na zagnieźdżenie. Kiedy kod ten zostaje wykonany, w tym samym czasie wykonywane są dwie pętle. Zewnętrzna przegląda listę kluczy, natomiast wewnętrzna przeszukuje listę obiektów dla każdego z kluczy. Zagnieźdżenie części `else` jest kwestią kluczową — jest ona wcięta na tę samą odległość co wiersz nagłówka wewnętrznej pętli `for`, dzięki czemu jest powiązana właśnie z nią (a nie z `if` czy zewnętrzną pętlą `for`).

Warto zauważyc, że przykład ten jest łatwiejszy do zapisania, jeżeli w teście przynależności wykorzystamy operator `in`. Ponieważ `in` w niejawnym sposobie przeszukuje obiekt, szukając dopasowania (przynajmniej logicznego), może zastąpić wewnętrzną pętlę.

```
>>> for key in tests:                                # Dla wszystkich kluczy
...     if key in items:                            # Niech Python sam sprawdzi
    dopasowanie
...         print(key, "znaleziono")
...     else:
...         print(key, "nie znaleziono!")
...
(4, 5) znaleziono
3.14 nie znaleziono!
```

Na ogół dobrze jest pozwolić Pythonowi na wykonanie jak największej ilości samodzielnej pracy — jak w powyższym rozwiążaniu — ze względu na zwięzłość i wydajność kodu.

Następny przykład jest podobny, ale zamiast wyświetlania wyników na ekranie tworzy listę do późniejszego użycia. Wykonuje typowe dla instrukcji `for` zadanie dotyczące struktur danych — zbieranie wspólnych elementów z dwóch sekwencji (ciągów znaków), co przypomina prostą procedurę wyznaczania części wspólnej zbiorów. Po wykonaniu pętli zmienna `res` będzie się odnosiła do listy zawierającej wszystkie elementy znalezione w łańcuchach `seq1` i `seq2`.

```
>>> seq1 = "mielonka"
>>> seq2 = "biedronka"
>>>
>>> res = []                                     # Na początek pusta lista
>>> for x in seq1:                            # Przeszukanie pierwszej
    sekwencji
...     if x in seq2:                          # Powtarzający się element?
...         res.append(x)                      # Dodanie na końcu listy wyników
...
>>> res
['i', 'e', 'o', 'n', 'k', 'a']
```

Niestety, kod ten będzie działał wyłącznie na dwóch określonych w nim zmiennych — `seq1` i `seq2`. Byłoby dobrze, gdyby tę pętlę można było w jakiś sposób uogólnić, tak by stała się narzędziem, z którego można skorzystać więcej niż tylko jeden raz. Jak zobaczymy, taki pomysł spowodował pojawienie się *funkcji* — koncepcji będącej tematem kolejnej części książki.

Przedstawiony kod wykorzystuje również klasyczny wzorzec *listy składanej* — zbieranie elementów listy w kolejnych iteracjach i z opcjonalnym filtrowaniem — i może być również zakodowany w bardziej zwięzły sposób:

```
>>> [x for x in seq1 if x in seq2] # Pozwól Pythonowi zbierać
wyniki
['i', 'e', 'o', 'n', 'k', 'a']
```

Aby poznać koniec tej historii, będziesz musiał jednak przeczytać jeszcze następny rozdział

Warto pamiętać – skanery plików

Pętle przydają się wszędzie tam, gdzie chcemy powtórzyć jakąś operację lub przetworzyć coś więcej niż jeden raz. Ponieważ pliki zawierają wiele znaków i wierszy, są jednym z bardziej typowych przypadków użycia dla pętli. By załadować całą zawartość pliku do łańcucha znaków za jednym razem, wystarczy wywołać metodę `read` obiektu pliku.

Aby jednak załadować plik w mniejszych częściach, często tworzymy albo pętlę while z instrukcją break na końcu pliku, albo pętlę for. Aby wczytać plik znak po znaku, przyda się jeden z poniższych fragmentów kodu.

```
file = open('test.txt')

while True:
    char = file.read(1)                      # Wczytanie znak po znaku
    if not char: break                       # Pusty ciąg znaków oznacza
                                                # koniec pliku
    print(char)

for char in open('test.txt').read():
    print(char)
```

Powyższa pętla for również przetwarza każdy znak po kolej, jednak ładuje cały plik do pamięci za jednym razem (i zakłada, że on się tam zmieści!). By zamiast tego wczytać plik wiersz po wierszu lub blok po bloku, można skorzystać z pętli while i jednego z poniższych rozwiązań.

```
file = open('test.txt')

while True:
    line = file.readline()                      # Wczytanie wiersz po wierszu
    if not line: break
    print(line.rstrip())                        # Wiersz zawiera już znak \n

file = open('test.txt', 'rb')

while True:
    chunk = file.read(10)                      # Wczytanie bajtowych
    fragmentów – do 10 bajtów
```

```
if not chunk: break
print(chunk)
```

Dane binarne zazwyczaj wczytuje się w blokach. Aby wczytać tekst *wiersz po wierszu*, łatwiej jest jednak napisać kod z pętlą `for`, którego dodatkową zaletą jest to, że będzie się on również szybciej wykonywał.

```
for line in open('test.txt').readlines():
    print(line.rstrip())
for line in open('test.txt'): # Użycie iteratorów – najlepszego trybu dla
    # tekstowych danych wejściowych
    print(line.rstrip())
```

Obie wersje działają zarówno w Pythonie 2.x, jak i 3.x. W pierwszym przykładzie użyto metody `readlines`, aby załadować wiersz po wierszu cały plik do listy łańcuchów znaków, a ostatni przykład wykorzystuje iteratory plików, które automatycznie odczytują po jednym wierszu w każdej iteracji pętli.

Ostatni z przykładów jest ogólnie najlepszym rozwiązaniem w przypadku plików tekstowych — jest prosty, działa dla dowolnie dużych plików i nie ładuje całego pliku naraz do pamięci. Wersja z iteratorem może być najszybsza, jednak wydajność operacji wejścia-wyjścia może się różnić w zależności od linii i wersji Pythona.

Wywołania metody `readlines` mogą być jednak nadal przydatne — na przykład do odwrócenia kolejności wierszy w pliku, przy założeniu, że jego zawartość mieści się w pamięci. Wbudowana metoda `reversed` akceptuje sekwencje, ale zezwala na używanie arbitralnie wybranych obiektów iterowalnych, które generują wartości — innymi słowy, lista działa, ale obiekt pliku już nie:

```
for line in reversed(open('test.txt').readlines()): ...
```

W kodzie napisanym w Pythonie 2.x można się spotkać z zastąpieniem nazwy `open` słowem `file`. Starsza metoda obiektu pliku `xreadlines` jest tam też wykorzystywana w celu uzyskania tego samego efektu, jaki daje automatyczny iterator wierszy pliku (metoda ta przypomina `readlines`, jednak nie ładuje całego pliku naraz do pamięci). Zarówno `file`, jak i `xreadlines` zostały usunięte w Pythonie 3.x, ponieważ są zbędne. Nie powinieneś ich także używać w nowym kodzie dla wersji 2.x, choć ciągle jeszcze mogą się pojawić w jakimś starszym kodzie czy zasobach.

Więcej informacji na temat przedstawionych rozwiązań znajdziesz w dokumentacji Pythona, a w rozdziale 14. będziemy zajmować się tematami związanymi z iteratorami plików. Zwrć również uwagę na ramkę „Warto pamiętać — polecenia powłoki i inne” w dalszej części rozdziału. Dodatkowe informacje na temat odczytywania plików znajdziesz również w rozdziale 37.; jak się przekonasz, pliki tekstowe i binarne mają w wersji 3.x nieco inną semantykę.

Techniki tworzenia pętli

Pętla `for`, którą opisywaliśmy przed chwilą, zalicza się do pętli wykorzystujących liczniki. Jest łatwiejsza do zapisania i szybsza do wykonania od pętli `while`, dlatego jest pierwszym narzędziem, po które powinniśmy sięgać, kiedy musimy przejść przez jakąś sekwencję lub inny obiekt iterowalny. Zasadniczo powinieneś opierać się *pokusie zliczania rzeczy w Pythonie* — jego narzędzia iteracyjne automatyzują większość zadań, które musisz wykonać, aby przeglądać kolekcje w językach niższego poziomu, takich jak C.

Istnieją jednak również sytuacje, w których będzie nam potrzebna bardziej wyspecjalizowana iteracja. Co należy na przykład zrobić, kiedy musimy przejść co drugi czy co trzeci element listy lub po drodze listę tę zmodyfikować? Co z przejściem większej liczby sekwencji równolegle, w tej samej pętli `for`? A co, jeżeli potrzebujesz również indeksów?

Tego typu unikalne iteracje można zawsze zapisać w kodzie z wykorzystaniem pętli `while` i ręcznego indeksowania, jednak Python udostępnia dwie funkcje wbudowane, które pozwalają na wykonanie wyspecjalizowanych iteracji za pomocą pętli `for`.

- Wbudowana funkcja `range` (dostępna w Pythonie od wersji 0.x) zwraca serię kolejnych, coraz większych liczb całkowitych, które mogą zostać wykorzystane jako indeksy dla pętli `for`.
- Wbudowana funkcja `zip` (dostępna w Pythonie od wersji 2.0) zwraca serię krotek równoległych elementów, która może zostać wykorzystana do przechodzenia wielu sekwencji w pętli `for`.
- Wbudowana funkcja `enumerate` (dostępna w Pythonie od wersji 2.3) generuje zarówno wartości, jak i indeksy elementów obiektu iterowalnego, więc nie musimy liczyć ręcznie.
- Wbudowana funkcja `map` (dostępna w Pythonie od wersji 1.0) może mieć podobny efekt jak `zip` w Pythonie 2.x, chociaż została usunięta w wersji 3.x.

Ponieważ pętle `for` zazwyczaj działają szybciej od pętli z licznikami opartych na `while`, użycie `for` w połączeniu z odpowiednimi narzędziami, takimi jak powyższe, zawsze będzie działało na naszą korzyść. Przyjrzyjmy się kolejno każdej z tych wbudowanych funkcji w kontekście typowych przypadków użycia. Jak zobaczymy, ich użycie może się nieznacznie różnić między wersjami 2.x a 3.x, a niektóre z nich są ważniejsze niż inne.

Pętle z licznikami – `range`

Nasza pierwsza funkcja związana z pętlami, `range`, jest tak naprawdę narzędziem ogólnego przeznaczenia, które można zastosować w wielu różnych kontekstach — spotkaliśmy się z nią już pokrótko w rozdziale 4. Choć najczęściej wykorzystywana jest do generowania indeksów dla pętli `for`, można jej użyć w każdym miejscu, w którym potrzebna jest nam lista liczb całkowitych. W Pythonie 2.x funkcja `range` tworzy listę; w Pythonie 3.x `range` jest *iteratorem* generującym elementy na żądanie, dlatego musimy opakować ją w wywołanie funkcji `list` w celu wyświetlenia wszystkich wyników naraz (więcej informacji na temat iteratorów znajdziesz w rozdziale 14.).

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

Z jednym argumentem wywołania, funkcja `range` generuje listę liczb całkowitych od zera do wartości argumentu (ale bez niej samej). Jeżeli przekażemy jej dwa argumenty, pierwszy uznawany jest za dolną granicę. Opcjonalny trzeci argument jest *krokiem*. Jeżeli go użyjesz, Python dodaje krok do każdej kolejnej liczby całkowitej wyniku (krok ma wartość domyślną 1). Zakresy mogą również być niedodatnie i nierośnace, o ile jest nam to do czegoś potrzebne.

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(range(5, -5, -1))
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Więcej szczegółowych informacji na temat takich obiektów iterowalnych znajdziesz w rozdziale 14. Zobaczysz tam również, że w Pythonie 2.x istnieje funkcja o nazwie `xrange`, która jest podobna do `range`, ale nie tworzy w pamięci listy wszystkich wyników od razu. Jest to

optymalizacja przestrzeni, która została uwzględniona w wersji 3.x przez zachowanie generatora w zakresie `range`.

Choć takie wyniki funkcji `range` mogą same w sobie być użyteczne, najbardziej przydają się wewnętrz pętli `for`. Po pierwsze, umożliwiają łatwe powtórzenie działania określonej liczbę razy. By na przykład wyświetlić trzy wiersze, wystarczy wykorzystać `range` do wygenerowania odpowiedniej ilości liczb całkowitych:

```
>>> for i in range(3):
...     print(i, 'Python')
...
0 Python
1 Python
2 Python
```

Pamiętaj, że w wersji 3.x pętle `for` automatycznie wymuszają wyniki z funkcji `range`, więc nie musimy jej opakowywać w wywołanie funkcji `list` (w wersji 2.x otrzymujemy listę tymczasową, chyba że zamiast tego wywołamy funkcję `xrange`).

Skanowanie sekwencji – pętla while z funkcją range kontra pętla for

Funkcja `range` jest często wykorzystywana do pośredniej iteracji po sekwencji, choć często nie jest to najlepsze rozwiązanie. Najłatwiejszym i najszybszym sposobem przejścia po sekwencji jest zawsze prosta pętla `for`, gdyż Python większość szczegółów zrobi za nas.

```
>>> X = 'mielonka'
>>> for item in X: print(item, end=' ')
# Prosta iteracja
...
m i e l o n k a
```

Wewnętrznie pętla `for` obsługuje szczegóły iteracji automatycznie, kiedy używa się jej w ten sposób. Jeżeli naprawdę musimy wziąć logikę indeksowania w swoje ręce, możemy to zrobić za pomocą pętli `while`.

```
>>> i = 0
>>> while i < len(X):
...     print(X[i], end=' ')
# Iteracja z pętlą while
...     i += 1
...
m i e l o n k a
```

W pętli `for` również możemy używać ręcznego indeksowania, jeżeli użyjemy funkcji `range` do wygenerowania listy indeksów, po których będziemy iterować. Jest to proces wieloetapowy, ale wystarczy wygenerować przesunięcia, a nie elementy z tych przesunięciach:

```
>>> X
'mielonka'
```

```

>>> len(X)                                # Długość łańcucha znaków
8
>>> list(range(len(X)))                  # Wszystkie poprawne
wartości przesunięcia dla X
[0, 1, 2, 3, 4, 5, 6, 7]
>>>
>>> for i in range(len(X)): print(X[i], end=' ')  # Ręczne indeksowanie za
pomocą range/len
...
m i e l o n k a

```

Warto zwrócić uwagę na to, że powyższy przykład przechodzi listę *wartości przesunięcia* dla X, a nie prawdziwych *elementów* X. Aby pobrać każdy z elementów, musimy z powrotem zindeksować X wewnętrz pętli. Jeżeli jednak wydaje Ci się to przesadą, to dlatego, że tak jest naprawdę: nie ma powodu, aby to robić w tak złożony sposób.

Ostatni omówiony przykład działa, jednak nie jest to najszybciej działające rozwiązanie. Wiąże się z nim również więcej pracy, niż jest to konieczne. O ile nie mamy specjalnych wymagań związanych z indeksowaniem, zwykle lepiej będzie użyć w Pythonie prostej postaci pętli `for`.

```

>>> for item in X: print(item, end=' ')      # Jeżeli możesz, korzystaj z
prostej formy iteracji

```

Zawsze, kiedy to możliwe, należy użyć `for` zamiast `while` i wywołanie `range` w pętli `for` należy traktować tylko jako ostatnią deskę ratunku. Prostsze rozwiązanie zawsze będzie lepsze. Nie zmienia to jednak w niczym faktu, że jak w przypadku każdej dobrej zasady istnieje od niej wiele wyjątków — o czym opowiemy w następnej sekcji.

Przetasowania sekwencji — funkcje `range` i `len`

Chociaż nie jest to idealne rozwiązanie do prostego skanowania sekwencji, wzorzec kodowania użyty w poprzednim przykładzie pozwala w razie potrzeby na bardziej wyspecjalizowane rodzaje przechodzenia przez sekwencje. Na przykład niektóre algorytmy mogą wykorzystywać zmianę kolejności elementów sekwencji — w celu generowania alternatyw w wyszukiwaniu, testowania efektów różnej kolejności wartości i tak dalej. Takie przypadki mogą wymagać przesunięć, aby rozdzielić sekwencję i złożyć ją z powrotem, jak to zostało pokazane w poniższym przykładzie; liczby całkowite zakresu zapewniają zliczanie powtórzeń w pierwszym przypadku i pozycję dla wycinka w drugim:

```

>>> S = 'mielonka'
>>> for i in range(len(S)):                  # Pętla wykonuje iteracje dla
wartości z zakresu 0..7
...   S = S[1:] + S[:1]                      # Przenosi pierwszy element na koniec
...   print(S, end=' ')
...
ielonkam elonkami lonkamie onkamiel nkamielo kamielon amielonk mielonka
>>> S
'mielonka'

```

```

>>> for i in range(len(S)):
...     X = S[i:] + S[:i]
...     print(X, end=' ')
...
mielonka ielonkam elonkami lonkamie onkamiel nkamielo kameleon amielonk

```

Jeżeli sposób działania tego kodu wydaje Ci się niezrozumiały, spróbuj najpierw przeanalizować pierwszą iterację. Druga działa w taki sam sposób jak pierwsza, z tym że wyświetla elementy w innej kolejności, ale nie zmienia pierwotnej wartości zmiennej. Ponieważ oba przykłady korzystają z wycinków, aby odpowiednio podzielić ciąg znaków, a potem go złączyć, będą działać również na dowolnych typach sekwencji i zwracać sekwencje tego samego typu, który jest przetwarzany — jeżeli używasz listy, tworzysz listę o zmienionej kolejności elementów:

```

>>> L = [1, 2, 3]
>>> for i in range(len(L)):
...     X = L[i:] + L[:i]
...     print(X, end=' ')
...
[1, 2, 3] [2, 3, 1] [3, 1, 2]

```

Wykorzystamy taki kod do testowania funkcji z różnymi kolejnościami argumentów w rozdziale 18. i rozszerzymy go na funkcje, generatory i tworzenie permutacji w rozdziale 20. — przekonasz się, że jest to bardzo przydatne narzędzie.

Przechodzenie niewyczerpujące — range kontra wycinki

Przykłady przedstawione w poprzedniej sekcji są poprawnymi zastosowaniami dla kombinacji funkcji `range/len`. Możemy również użyć tej techniki do pomijania wybranych elementów sekwencji:

```

>>> S = 'abcdefghijklk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]
>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k

```

W tym przykładzie przetwarzamy *co drugi* element łańcucha `S` dzięki przechodzeniu listy wygenerowanej przez funkcję `range`. Aby przetworzyć co trzeci element, wystarczy zmienić trzeci argument funkcji `range` na 3. W rezultacie takie wykorzystanie funkcji `range` pozwala na przeskakiwanie elementów sekwencji w pętlach przy jednoczesnym zachowaniu prostoty związanej z konstrukcją pętli `for`.

Mimo to, w większości przypadków również nie jest to obecnie najbardziej poleczana technika w Pythonie. Jeśli naprawdę chcesz pominąć niektóre elementy sekwencji, rozszerzona forma *wyrażenia z wycinkiem* z trzema granicami (zaprezentowana w rozdziale 7.) umożliwia

osiągnięcie tego samego w łatwiejszy sposób. Aby przetworzyć co drugi znak z łańcucha S, wystarczy wykonać wycinek z krokiem o wartości 2.

```
>>> S = 'abcdefghijklk'  
>>> for c in S[::2]: print(c, end=' ')  
...  
a c e g i k
```

Rezultat jest identyczny, ale sam kod jest znacznie łatwiejszy do napisania i przeanalizowania przez innych użytkowników. Potencjalną zaletą użycia tutaj funkcji range jest miejsce: wycinki tworzą kopię łańcucha znaków zarówno w wersji 2.x, jak i 3.x, podczas gdy funkcja range w 3.x i xrange w 2.x nie tworzą listy; w przypadku bardzo dużych ciągów znaków może to oszczędzić nieco miejsca w pamięci.

Modyfikowanie list – range kontra listy składane

Inną często spotykaną sytuacją, w której można użyć kombinacji funkcji range/len, jest pętla for modyfikująca elementy przetwarzanej listy. Założymy na przykład, że z jakiegoś powodu musimy dodać 1 do każdego elementu listy. Można spróbować tego dokonać za pomocą prostej pętli for, jednak wynik niekoniecznie będzie taki, jak byśmy chcieli.

```
>>> L = [1, 2, 3, 4, 5]  
>>> for x in L:  
...     x += 1  
...  
>>> L  
[1, 2, 3, 4, 5]  
>>> x  
6
```

Takie rozwiązanie nie działa — modyfikowana jest zmienna pętli x, a nie lista L. Przyczyny takiego stanu rzeczy są dość subtelne. Przy każdym przejściu pętli zmienna x odnosi się do kolejnej liczby całkowitej pobranej z listy. W pierwszej iteracji x jest na przykład liczbą całkowitą 1. W kolejnej iteracji ciało pętli ustawia x na inny obiekt — liczbę całkowitą 2 — jednak nie uaktualnia w miejscu listy, z której oryginalnie pochodziła liczba 1.

By naprawdę zmodyfikować listę w czasie jej przechodzenia, musimy użyć indeksów, które pozwolą na przypisanie uaktualnionej wartości do każdej pozycji listy w miarę jej przechodzenia. Kombinacja funkcji range i len pozwala uzyskać pożądane indeksy.

```
>>> L = [1, 2, 3, 4, 5]  
>>> for i in range(len(L)):  
...     L[i] += 1  
...  
>>> L  
[2, 3, 4, 5, 6]
```

Taki kod pozwala na modyfikację listy w miarę przechodzenia pętli. Nie da się uzyskać tego samego za pomocą prostej pętli w stylu `for x in L:`, ponieważ taka pętla przechodzi same elementy listy, a nie ich pozycje. Co jednak z odpowiednikiem tego rozwiązania wykorzystującym pętlę `while`? Taka pętla będzie wymagać od nas nieco więcej pracy i naprawdopodobniej będzie również działała wolniej w zależności od wersji Pythona (tak dzieje się w wersjach 2.7 i 3.3, choć w tej ostatniej nieco mniej wolno — zobaczymy, jak to sprawdzić, w rozdziale 21.):

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

I tutaj rozwiązanie z funkcją `range` może nie być idealne. Wyrażenie listy składanej w postaci:

```
[x + 1 for x in L]
```

prawdopodobnie będzie działać dzisiaj szybciej i wykonałoby podobną pracę, choć nie zmieniłoby oryginalnej listy w miejscu (moglibyśmy przypisać nowy obiekt listy z wyrażenia z powrotem do `L`, jednak nie uaktualniłoby to innych referencji do oryginalnej listy). Ponieważ listy składane są kluczową koncepcją związaną z pętlami, zostawimy pełne omówienie tego zagadnienia do następnego rozdziału.

Przechodzenie równoległe — `zip` oraz `map`

Nasza następna technika kodowania pętli rozszerza zakres pętli. Jak widzieliśmy, wbudowana funkcja `range` pozwala na przechodzenie sekwencji za pomocą pętli `for` w sposób niewyczerpujący. W podobny sposób wbudowana funkcja `zip` pozwala na wykorzystanie pętli `for` do równoległego przejścia większej liczby sekwencji — w tej samej pętli, ale bez nakładania się na siebie w czasie. W prostej operacji funkcja ta przyjmuje jako argument jedną lub większą liczbę sekwencji i zwraca serię krotek łączących w pary równolegle elementy z tych sekwencji. Założymy na przykład, że pracujemy z dwoma listami (może to być na przykład lista nazwisk i adresów sparowanych według pozycji):

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
```

Aby połączyć elementy z tych dwóch list, możemy wykorzystać funkcję `zip` do utworzenia listy par krotek. Tak jak `range`, `zip` jest w wersji 2.x listą, ale w Pythonie 3.x obiektem iterowalnym, dlatego musimy opakować tę funkcję w wywołanie `list` w celu wyświetlenia wszystkich jej wyników naraz — więcej informacji o iteratorach znajdziesz w kolejnym rozdziale.

```
>>> zip(L1,L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))                                # list() wymagane w Pythonie
3.x, w 2.x nie
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Taki wynik może być przydatny również w innych kontekstach, natomiast w połączeniu z `for` obsługuje iteracje równoległe.

```
>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Powyżej przechodzimy przez wyniki wywołania funkcji `zip`, czyli pary elementów pobranych z dwóch list. Warto zauważyc, że pętla `for` ponownie wykorzystuje przypisanie krotek (w postaci, z jaką spotkaliśmy się już wcześniej) w celu rozpakowania każdej krotki w wyniku `zip`. Za pierwszym razem wygląda to tak, jakbyśmy wykonali instrukcję przypisania $(x, y) = (1, 5)$.

Rezultat będzie taki, że w pętli przechodzimy zarówno listę `L1`, jak i listę `L2`. Podobny efekt moglibyśmy osiągnąć za pomocą pętli `while`, która obsługuje ręczne indeksowanie, jednak wymagałoby to więcej kodu i prawdopodobnie działałoby wolniej od rozwiązania z `for` oraz `zip`.

Funkcja `zip` jest tak naprawdę bardziej uniwersalna, niż mogłoby to wynikać z powyższego przykładu. Przyjmuje ona dowolny typ sekwencji (a tak naprawdę obiekt, na którym można wykonać iterację — w tym pliki) i może przyjąć więcej niż dwa argumenty. W przypadku trzech argumentów, jak w poniższym przykładzie, buduje listę krotek trzyelementowych z elementami z każdej z sekwencji, odzwierciedlając w ten sposób kolumny (otrzymujemy krotkę N -argumentową dla N argumentów).

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))                                # Trzy krotki dla trzech
argumentów
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Co więcej, kiedy długość argumentów różni się, funkcja `zip` odcina wynikowe krotki do długości najkrótszej sekwencji. W poniższym przykładzie łączymy ze sobą dwa łańcuchy w celu równoległego wybrania znaków, jednak wynik składa się jedynie z tylu krotek, jaką długość ma najkrótsza sekwencja:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))                                    # Odcina krotki do długości
najkrótszego elementu
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Równoznaczność funkcji map w Pythonie 2.x

W Pythonie 2.x powiązana z `zip` funkcja wbudowana `map` w podobny sposób łączy w pary elementy sekwencji, jednak dopełnia krótsze sekwencje za pomocą obiektu `None`, jeśli długość argumentów jest różna, zamiast odcinać wynik do długości krótszej sekwencji:

```

>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>> map(None, S1, S2)          # Tylko w Pythonie 2.x: dopełnia do
                               # długości najdłuższego elementu
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]

```

Powyższy przykład używa zdegenerowanej formy funkcji `map`, która w Pythonie 3.x nie jest już obsługiwana. Normalnie `map` przyjmuje w postaci argumentów funkcję oraz jedną lub większą liczbę sekwencji i zbiera wynik wywołania funkcji z równoległymi elementami pobranymi z sekwencji.

Funkcję `map` będziemy omawiać bardziej szczegółowo w rozdziałach 19. oraz 20., natomiast poniższy przykład odwzorowuje wbudowaną funkcję `ord` na każdy element łańcucha znaków i zbiera wyniki. Tak jak `zip`, funkcja `map` jest w Pythonie 3.x generatorem wartości, dlatego konieczne jest przekazanie jej do `list` w celu zebrania wszystkich wyników naraz.

```

>>> list(map(ord, 'mielonka'))
[109, 105, 101, 108, 111, 110, 107, 97]

```

Kod ten działa tak samo jak poniższa instrukcja pętli, jednak często funkcja `map` jest szybsza (jak zobaczymy w rozdziale 21.):

```

>>> res = []
>>> for c in 'mielonka': res.append(ord(c))
>>> res
[109, 105, 101, 108, 111, 110, 107, 97]

```



Uwaga na temat wersji: Zdegenerowana postać `map` wykorzystująca argument funkcji `None` nie jest już obsługiwana w Pythonie 3.x, ponieważ w dużej mierze pokrywa się z `zip` (a do tego, szczerze mówiąc, nieco odbiegała od celu zastosowania funkcji `map`). W wersji 3.x należy albo skorzystać z funkcji `zip`, albo samodzielnie napisać kod pętli dopełniający wyniki. Zobaczmy, jak można to zrobić, w rozdziale 20., po tym, jak będziemy mieli okazję zapoznać się z pewnymi dodatkowymi zagadnieniami związanymi z iteracją.

Tworzenie słowników za pomocą funkcji `zip`

Przyjrzyjmy się kolejnemu zastosowaniu funkcji `zip`. W rozdziale 8. zasugerowałem, że zaprezentowane wyżej wywołanie `zip` może się również przydać do generowania słowników, kiedy zbiory kluczy i wartości muszą być obliczane w czasie wykonania. Skoro już zapoznaliśmy się z funkcją `zip`, czas wyjaśnić, co ma ona wspólnego z tworzeniem słowników. Jak powiedzieliśmy wcześniej, słownik zawsze można utworzyć za pomocą literała lub przypisania wartości do kluczy.

```

>>> D1 = {'mielonka':1, 'jajka':3, 'tost':5}
>>> D1
{'jajka': 3, 'tost': 5, 'mielonka': 1}
>>> D1 = {}
>>> D1['mielonka'] = 1
>>> D1['jajka'] = 3

```

```
>>> D1['tost'] = 5
```

Co jednak zrobić, kiedy program uzyskuje klucze i wartości słownika w postaci *list* w czasie działania, już po utworzeniu samego skryptu? Powiedzmy na przykład, że mamy poniższe listy kluczy i wartości, pobrane od użytkownika, odczytane z pliku czy po prostu pozyskane z innego, dynamicznego źródła danych:

```
>>> keys = ['mielonka', 'jajka', 'tost']
>>> vals = [1, 3, 5]
```

Jedną z możliwości przekształcenia ich w słownik będzie zastosowanie na listach funkcji *zip* i równolegle przejście ich za pomocą pętli *for*.

```
>>> list(zip(keys, vals))
[('mielonka', 1), ('jajka', 3), ('tost', 5)]
>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'jajka': 3, 'tost': 5, 'mielonka': 1}
```

Okazuje się jednak, że od wersji 2.2 można całkowicie pominąć pętlę *for* i po prostu przekazać połączone w pary za pomocą *zip* listy kluczy i wartości do wywołania wbudowanego konstruktora słownika *dict*.

```
>>> keys = ['mielonka', 'jajka', 'tost']
>>> vals = [1, 3, 5]
>>> D3 = dict(zip(keys, vals))
>>> D3
{'jajka': 3, 'tost': 5, 'mielonka': 1}
```

Wbudowana nazwa *dict* jest tak naprawdę nazwą typu w Pythonie (o nazwach typów i ich podklassach dowiemy się więcej w rozdziale 32.). Jej wywołanie powoduje coś w stylu konwersji z listy na słownik, choć tak naprawdę jest to żądanie konstrukcji obiektu.

W kolejnym rozdziale zapoznamy się z podobną, jednak bogatszą koncepcją *list składanych*, które budują listy w jednym wyrażeniu. Powrócimy także do *słowników składanych* z Pythona 3.x i 2.7, stanowiących alternatywę dla wywołania konstruktora *dict* dla połączonych za pomocą *zip* par kluczy i wartości.

```
>>> {k: v for (k, v) in zip(keys, vals)}
{'jajka': 3, 'tost': 5, 'mielonka': 1}
```

Generowanie wartości przesunięcia i elementów — *enumerate*

Nasza funkcja pomocnicza z ostatniej pętli została zaprojektowana do obsługi dwóch trybów użycia. Wcześniej omawialiśmy wykorzystanie funkcji *range* do wygenerowania wartości przesunięcia elementów łańcucha znaków, a nie samych elementów znajdujących się na tych pozycjach. W niektórych programach potrzebne nam będą jednak obie wartości — używany

element i przy okazji jego wartość przesunięcia. Zazwyczaj takie coś rozwiązywano za pomocą prostej pętli `for`, która przechowywała również licznik bieżącej wartości przesunięcia elementu.

```
>>> S = 'mielonka'  
>>> offset = 0  
>>> for item in S:  
...     print(item, 'występuje na pozycji przesunięcia', offset)  
...     offset += 1  
  
...  
m występuje na pozycji przesunięcia 0  
i występuje na pozycji przesunięcia 1  
e występuje na pozycji przesunięcia 2  
l występuje na pozycji przesunięcia 3  
o występuje na pozycji przesunięcia 4  
n występuje na pozycji przesunięcia 5  
k występuje na pozycji przesunięcia 6  
a występuje na pozycji przesunięcia 7
```

To rozwiązanie działa, jednak we wszystkich nowych wersjach Pythona 2.x i 3.x (począwszy od 2.3) to samo robi za nas nowa funkcja wbudowana `enumerate` — jej efektem netto jest nadanie pętlom licznika „za darmo”, bez rezygnacji z prostoty automatycznej iteracji:

```
>>> S = 'mielonka'  
>>> for (offset, item) in enumerate(S):  
...     print(item, 'występuje na pozycji przesunięcia', offset)  
  
...  
m występuje na pozycji przesunięcia 0  
i występuje na pozycji przesunięcia 1  
e występuje na pozycji przesunięcia 2  
l występuje na pozycji przesunięcia 3  
o występuje na pozycji przesunięcia 4  
n występuje na pozycji przesunięcia 5  
k występuje na pozycji przesunięcia 6  
a występuje na pozycji przesunięcia 7
```

Funkcja `enumerate` zwraca *obiekt generatora* — rodzaj obiektu obsługujący protokół iteracji, z którym zapoznamy się w następnym rozdziale i który omówimy bardziej szczegółowo w kolejnej części książki. W skrócie, obsługuje on wbudowaną metodę `next` i zwracającą za każdym przejściem pętli krotkę `(indeks, wartość)`. Pętla `for` automatycznie przechodzi przez te krotki, co pozwala nam rozpakowywać ich wartości, podobnie jak to robiliśmy w przypadku funkcji `zip`.

```
>>> E = enumerate(S)
>>> E
<enumerate object at x0000000002A8B900>
>>> next(E)
(0, 'm')
>>> next(E)
(1, 'i')
>>> next(E)
(2, 'e')
```

Jak zawsze, normalnie nie widzimy tego wszystkiego, ponieważ konteksty iteracyjne — w tym listy składane, temat rozdziału 14. — wykonują protokół iteracji automatycznie.

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'i', 'ee', 'lll', 'ooo', 'nnnnn', 'kkkkkk', 'aaaaaaaa']
>>> for (i, l) in enumerate(open('test.txt')):
...     print('%s' % (i, l.rstrip()))
...
0) aaaaaa
1) bbbbbbb
2) ccccccc
```

Aby w pełni zrozumieć zagadnienia związane z iteracją, takie jak `enumerate`, `zip` czy listy składane, musimy przejść do kolejnego rozdziału zawierającego ich omówienie z formalnego punktu widzenia.

Warto pamiętać — polecenia powłoki i inne

W poprzedniej ramce omawialiśmy zastosowanie pętli do przetwarzania plików. Jak krótko wspominaliśmy w rozdziale 9., wywołanie metody `os.popen` również udostępnia interfejs plikowy pozwalający na odczytywanie wyników działania polecen powłoki wywoływanych z poziomu Pythona. Teraz gdy zakończyliśmy już omawianie pętli, chciałby zaprezentować przykład zastosowania tego narzędzia w akcji — aby z poziomu Pythona uruchomić wybrane polecenie powłoki i odczytać wyniki jego działania, musimy przekazać nazwę polecenia w postaci łańcucha znaków do metody `os.popen` i odczytać wyniki z plikopodobnego obiektu, który ta metoda zwraca (jeżeli będziesz miał na swoim komputerze problem z kodowaniem Unicode w wyświetlanych wynikach, powinieneś zatrzymać się na rozdziale 25.):

```
>>> import os
>>> F = os.popen('dir')                                # Odczytywanie wiersz po
wierszu
>>> F.readline()
' Volume in drive C has no label.\n'
>>> F = os.popen('dir')                                # Odczytywanie w blokach o
podanym rozmiarze
```

```

>>> F.read(50)
' Volume in drive C has no label.\n Volume Serial Nu'
>>> os.popen('dir').readlines()[0]           # Odczytywanie wszystkich
wierszy: indeks
' Volume in drive C has no label.\n'
>>> os.popen('dir').read():50                # Odczytywanie wszystkiego od
razu: wycinek
' Volume in drive C has no label.\n Volume Serial Nu'
>>> for line in os.popen('dir'):             # Pętla iteratora pliku
...     print(line.rstrip())
...
Volume in drive C has no label.
Volume Serial Number is D093-D1F7
...i tak dalej...

```

Spowoduje to uruchomienie polecenia dir w systemie Windows, ale w ten sposób można wywołać dowolny inny program, który da się uruchomić z poziomu wiersza polecenia. Możemy użyć tego schematu na przykład do wyświetlenia wyniku działania polecenia systeminfo systemu Windows — metoda os.system po prostu uruchamia polecenie powłoki, ale os.popen podłącza się również do jego strumieni danych; oba poniższe przykłady pokazują dane wyjściowe polecenia powłoki w prostym oknie konsoli, ale w pierwszym przypadku wyniki działania mogą nie być wyświetlane w graficznych interfejsach użytkownika, takich jak IDLE:

```

>>> os.system('systeminfo')
...wyniki są wyświetlane w konsoli tekstowej, środowisko IDLE wywołuje
osobne okno...
0
>>> for line in os.popen('systeminfo'): print(line.rstrip())
Host Name: MARK-VAIO
OS Name: Microsoft Windows 7 Professional
OS Version: 6.1.7601 Service Pack 1 Build 7601
...tutaj pojawi się dużo informacji o systemie...

```

Gdy otrzymamy wyniki działania polecenia w formie tekstopowej, możemy użyć dowolnego narzędzia lub techniki przetwarzania tekstów, w tym formatowania wyświetlania i parsowania zawartości:

```

# Wyświetlane wyniki są formatowane i ograniczone
>>> for (i, line) in enumerate(os.popen('systeminfo')):
...     if i == 4: break
...     print('%05d %s' % (i, line.rstrip()))
...

```

```
00000)
00001)                               Host Name: MARK-VAIO
00002)                               OS Name: Microsoft Windows 7 Professional
00003)                               OS Version: 6.1.7601 Service Pack 1 Build
7601

# Parsowanie wybranych wierszy niezależne od wielkości liter
>>> for line in os.popen('systeminfo'):
...     parts = line.split(':')
...     if parts and parts[0].lower() == 'system type':
...         print(parts[1].strip())
...
x64-based PC
```

Metodę os.popen ponownie zobaczymy w akcji w rozdziale 21., gdzie wdrożymy ją, aby odczytać wyniki działania złożonego polecenia wywoływanego z poziomu wiersza poleceń, oraz w rozdziale 25., gdzie zostanie użyta do porównania wyników testowanych skryptów.

Narzędzia takie jak `os.popen` i `os.system` (oraz niepokazany tutaj moduł `subprocess`) pozwalają na wykorzystanie wiersza poleceń dowolnej konsoli na Twoim komputerze, ale możesz także pisać emulatory przetwarzające wyniki w czasie działania procesu. Na przykład symulacja jednej z możliwości uniksowego narzędzia `awk`, pozwalającej na usuwanie kolumn z plików tekstowych, jest w Pythonie niemal banalna i może być funkcja wielokrotnego użytku w tym procesie:

```
# emulacja awk: wyodrębnienie siódmej kolumny z pliku, gdzie poszczególne
# pola są rozdzielone białymi znakami

for val in [line.split()[6] for line in open('input.txt')]:
    print(val)

# To samo, ale z ulepszonym kodem, który zachowuje wyniki

col7 = []

for line in open('input.txt'):
    cols = line.split()
    col7.append(cols[6])

for item in col7: print(item)

# To samo, ale w postaci funkcji wielokrotnego użytku (patrz kolejna część
# książki)

def awker(file, col):
    return [line.rstrip().split()[col-1] for line in open(file)]

print(awker('input.txt', 7))                      # Lista ciągów znaków

print(','.join(awker('input.txt', 7)))            # Dodaje przecinki pomiędzy
# elementami
```

Python zapewnia analogiczny, plikopodobny dostęp do szerokiej gamy danych — w tym do tekstu zwracanego przez *strony internetowe* identyfikowane przez adres URL; więcej szczegółowych informacji na ten temat znajdziesz w części V naszej książki, gdzie poruszymy między innymi zagadnienia związane z pakietem, który tutaj importujemy, oraz innymi zasobami i narzędziami tego typu (takie rozwiązania działają na przykład w wersji 2.x, ale zamiast metody `urllib.request` musimy użyć metody `urllib`, która zwraca wyniki w postaci ciągów tekstu):

```
>>> from urllib.request import urlopen  
>>> for line in urlopen('http://learning-python.com/books'):  
...     print(line)  
...  
b'<HTML>\n'  
b'\n'  
b'<HEAD>\n'  
b"<TITLE>Mark Lutz's Book Support Site</TITLE>\n"  
...itd....
```

Podsumowanie rozdziału

W niniejszym rozdziale zapoznaliśmy się z instrukcjami pętli Pythona, a także z pewnymi koncepcjami związanymi z pętlami w tym języku. Przyjrzaliśmy się instrukcjom `while` i `for`, a także zobaczyliśmy, jak działają powiązane z nimi części `else`. Omówiliśmy również instrukcje `break` i `continue`, które mają znaczenie jedynie wewnętrz pętli, a także poznaliśmy kilka wbudowanych narzędzi wykorzystywanych często w pętlach `for` — w tym `range`, `zip`, `map` oraz `enumerate` (choć niektóre szczegóły ich działania jako iteratorów w Pythonie 3.x zostały celowo skrócone bądź pominięte).

W kolejnym rozdziale będziemy kontynuowali omawianie iteracji, przedstawiając listy składane oraz protokół iteracji Pythona — zagadnienia ściśle powiązane z pętlami `for`. Wyjaśnimy tam także pewne subtelności narzędzi iteracyjnych, które zostały wprowadzone w niniejszym rozdziale, takich jak `range` oraz `zip`. Jak zawsze jednak najpierw należy za pomocą quizu przećwiczyć kwestie przedstawione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Jakie są najważniejsze różnice funkcjonalne pomiędzy `while` i `for`?
2. Jaka jest różnica pomiędzy `break` i `continue`?
3. Kiedy w pętli wykonywana jest część `else`?
4. W jaki sposób można w Pythonie zapisać pętlę opartą na liczniku?
5. Do czego w pętli `for` można wykorzystać funkcję `range`?

Sprawdź swoją wiedzę — odpowiedzi

1. Pętla `while` jest ogólną instrukcją pętli, jednak to `for` zaprojektowane jest z myślą o iteracji wykonywanej na elementach sekwencji lub innego obiektu iterowalnego. Choć pętla `while` może imitować `for` w przypadku pętli opartych na licznikach, wymaga większej ilości kodu i może działać wolniej.
2. Instrukcja `break` powoduje natychmiastowe wyjście z pętli (znajdziemy się poniżej całej instrukcji pętli `while` lub `for`), natomiast `continue` przeskakuje z powrotem na górę pętli (znajdziemy się tuż przed testem w `while` lub kolejnym elementem pobieranym w `for`).
3. Część `else` w pętlach `while` i `for` zostanie wykonana raz, kiedy pętla się kończy — o ile kończy się normalnie, bez trafienia na instrukcję `break`. Instrukcja `break` powoduje natychmiastowe wyjście z pętli i pominięcie części `else` (o ile jest ona w ogóle obecna).
4. Pętle liczników można zapisywać za pomocą instrukcji `while`, która ręcznie przechowuje indeks, lub za pomocą pętli `for`, która do generowania kolejnych wartości przesunięcia wykorzysta wbudowaną funkcję `range`. Żadna z nich nie jest preferowanym sposobem, jeśli po prostu potrzebujemy przejść wszystkie elementy sekwencji. W takiej sytuacji należy zamiast tego, kiedy tylko jest to możliwe, użyć prostej instrukcji `for`, bez `range` i liczników. Jest ona łatwiejsza do zapisania w kodzie, a także zazwyczaj szybsza do wykonania.
5. Funkcję wbudowaną `range` można wykorzystać w pętli `for` w celu zaimplementowania ustalonej liczby powtórzeń, przejście po wartościach przesunięć zamiast po elementach znajdujących się na tych pozycjach, pominięcia kolejnych elementów w miarę przechodzenia, a także modyfikacji listy w trakcie przechodzenia jej. Żadna z tych ról nie wymaga `range` i w większości przypadków istnieją alternatywy — przejście samych elementów, wycinki z trzema wartościami granicznymi, a także listy składane są obecnie nieraz lepszymi rozwiązaniami (pomimo naturalnych upodobań byłych programistów języka C, którzy chcą wszystko zliczać).

Rozdział 14. Iteracje i listy składane

W poprzednim rozdziale poznaliśmy dwie instrukcje Pythona odpowiedzialne za obsługę pętli: `while` i `for`. Pętle te są w stanie obsłużyć większość powtarzalnych zadań, ale do pracy wymagają danych sekwencyjnych, które są zjawiskiem tak powszechnym, że Python dorobił się narzędzi upraszczających i przyspieszających tego typu operacje. Niniejszy rozdział rozpoczyna naszą eksplorację tych narzędzi. A dokładniej: zajmiemy się koncepcją *protookołu iteracyjnego* zaimplementowanego w Pythonie (model oparty na wywołaniu metod obiektów iteracyjnych), jak również poznamy podstawy konstrukcji *list składanych* — mechanizmu blisko spokrewnionego z pętlą `for`, pozwalającego na wykonywanie wyrażeń na sekwencji elementów obiektu iterowalnego.

Obydwa wspomniane narzędzia są ściśle związane zarówno z pętlami, jak i z funkcjami, więc zajmiemy się nimi w kilku turach:

- W tym rozdziale omówimy podstawowe zagadnienia w kontekście pętli, co będzie niejako kontynuacją poprzedniego rozdziału.
- W rozdziale 20. omówimy je w kontekście narzędzi opartych na funkcjach i rozszerzymy ten temat o *generatory* wbudowane i zdefiniowane przez użytkownika.
- W rozdziale 30. zamieszczać będziemy podsumowanie tej historii, w którym dowiemy się o obiektach iterowalnych zdefiniowanych przez użytkownika i zakodowanych za pomocą *klas*.

W tym rozdziale zapoznamy się również z dodatkowymi narzędziami iteracyjnymi w Pythonie i nowymi obiektami iterowalnymi dostępnymi w Pythonie 3.x — gdzie obiekty iterowalne stają się jeszcze bardziej wszechobecne.

Z góry jednak uprzedzamy — część zagadnień poruszanych w tym rozdziale może na pierwszy rzut oka wydać się dość skomplikowana, jednak w praktyce okazują się bardzo użyteczne i dają mnóstwo możliwości. Ich znajomość nie jest ściśle wymagana, ale ponieważ stały się one niezwykle powszechnie w programach w Pythonie, zrozumienie tej tematyki znaczco pomoże w analizie kodu napisanego przez innych programistów.

Iteracje — pierwsze spojrzenie

W poprzednim rozdziale wspomniałem, że pętla `for` może działać na dowolnym typie sekwencji Pythona, w tym na listach, krotkach i łańcuchach znaków — jak w poniższym przykładzie.

```
>>> for x in [1, 2, 3, 4]: (print x ** 2, end=' ')
2.x: print x**2
...
1 4 9 16
>>> for x in (1, 2, 3, 4): (print x ** 3, end=' ')
# W wersji
```

```
...
1 8 27 64
>>> for x in 'mielonka': (print x * 2, end=' ')
...
mm ii ee ll oo nn kk aa
```

Tak naprawdę pętla `for` okazuje się o wiele bardziej uniwersalna, niż wynikałoby to z powyższych przykładów. Działa ona bowiem na dowolnym *obiekcie iterowalnym*. Tak samo jest w przypadku wszystkich innych narzędzi iteracyjnych w Pythonie, przechodzących obiekty od lewej do prawej strony — w tym pętli `for`, list składanych, testów przynależności `in`, a także wbudowanej funkcji `map`.

Koncepcja obiektów iterowalnych (ang. *iterable object*) jest w Pythonie czymś relatywnie nowym, ale stała się cechą dominującą tego języka. Jest to w zasadzie uogólnienie pojęcia sekwencji. Obiekt uznawany jest za taki, jeśli jest albo fizycznie przechowywaną sekwencją, albo obiektem zwracającym jeden wynik naraz w kontekście narzędzia iteracyjnego, takiego jak pętla `for`. W pewnym sensie obiekty tego typu obejmują zarówno fizyczne sekwencje, jak i *sekwencje wirtualne* tworzone na żądanie.



Terminologia dotycząca tej tematyki jest zdefiniowana dość luźno. W tekście posługuję się terminami „obiekt iterowalny” oraz „iterator” w odniesieniu do ogólnie rozumianych obiektów obsługujących mechanizm iteracji. Aby uniknąć niepotrzebnych nieporozumień, przypominam, że używane w tej książce określenie *obiekt iterowalny* odnosi się do obiektu, który obsługuje wywoływanie metody `iter()`, natomiast określenie *iterator* odnosi się do obiektu zwracanego przez funkcję `iter()`, obsługującego metodę `next(I)`. Sposoby wywoływania obu metod omówimy już niebawem.

Konwencja ta nie jest jednak uniwersalnie stosowana ani w świecie Pythona, ani w tej książce; określenie „iterator” jest czasem używane także w przypadku narzędzi, które iterują. Rozdział 20. rozszerza tę kategorię o termin „generator”, który odnosi się do obiektów automatycznie obsługujących protokół iteracji, a zatem iterowalnych — choć przecież wszystkie obiekty iterowalne generują wyniki!

Protokół iteracyjny — iteratory plików

Jednym z łatwiejszych sposobów zrozumienia, co to może oznaczać, jest przyjrzenie się, jak to działa w przypadku typu wbudowanego, takiego jak plik. W przykładach omawianych w tym rozdziale będziemy używać następującego pliku wejściowego:

```
>>> print(open('script2.py').read())
import sys
print(sys.path)
x = 2
print(x ** 32)
>>> open('script2.py').read()
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'
```

Jak pamiętasz z rozdziału 9., otwarte obiekty plików obsługują metodę o nazwie `readline()`, wczytującą po jednym wierszu tekstu z pliku na raz. Przy każdym wywołaniu metody

`readline()` przesuwamy się do kolejnego wiersza. Na końcu pliku zwracany jest pusty łańcuch znaków, który możemy wykryć w celu wyjścia z pętli.

```
>>> f = open('script2.py')      # Odczytanie 4-wierszowego skryptu z bieżącego katalogu
>>> f.readline()              # funkcja readline wczytuje po jednym wierszu
'import sys\n'
>>> f.readline()
'print sys.path\n'
>>> f.readline()
'x = 2\n'
>>> f.readline()              # Ostatni wiersz może, ale nie musi być zakończony znakiem \n
'print 2 ** 33\n'
>>> f.readline()              # Zwraca pusty ciąg znaków na końcu pliku
''
```

Pliki obsługują również metodę o nazwie `__next__` (w wersji 3.x) i `next` (w wersji 2.x), która działa dokładnie w ten sam sposób — przy każdym wywołaniu zwraca kolejny wiersz z pliku. Jedyną zauważalną różnicą jest to, że metoda `__next__()` na końcu pliku w miejsce pustego łańcucha znaków zwraca wyjątek `StopIteration`.

```
>>> f = open('script2.py')      # __next__ również wczytuje po jednym wierszu
>>> f.__next__()               # ale wywołuje wyjątek na końcu pliku
'import sys\n'
>>> f.__next__()               # Użyj f.next() w wersji 2.x lub next(f) w wersji 2.x lub 3.x
'print (sys.path)\n'
>>> f.__next__()
'x = 2\n'
>>> f.__next__()
'print 2 ** 32\n'
>>> f.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Ten interfejs jest mniej więcej tym, co w Pythonie nazywamy *protokołem iteracji*. Dowolny obiekt obsługujący metodę `__next__()` przechodzącą do kolejnego wyniku, który zgłasza wyjątek `StopIteration` na końcu serii wyników, nazywamy w Pythonie iteratorem. Takie obiekty można przetwarzać za pomocą pętli `for` czy innego narzędzia iteracyjnego, ponieważ wszystkie narzędzia iteracyjne wewnętrznie działają dzięki wywoływaniu metody `__next__()` w każdej iteracji i przechwytywaniu wyjątku `StopIteration` w celu ustalenia, kiedy skończyć. Jak

zobaczmy za chwilę, w przypadku niektórych obiektów pełny protokół zawiera dodatkowy krok do wywołania metody `iter`, ale nie jest to wymagane w przypadku plików.

Rezultat tej magii jest taki, że — jak wspominaliśmy w rozdziałach 9. i 13. — najlepszym sposobem na odczytywanie pliku tekstowego wiersz po wierszu jest obecnie *nieodczytywanie go wcale*. Zamiast tego lepiej jest pozwolić pętli `for` na automatyczne wywołanie metody `__next__()` w celu przejścia do kolejnego wiersza z każdą iteracją. Poniższy kod wczytuje plik wiersz po wierszu (wyświetlając przy okazji wersję zapisaną wielkimi literami) bez jawnego odczytywania czegokolwiek z pliku.

```
>>> for line in open('script2.py'):          # Użycie iteratora do
   odczytywania pliku wiersz po wierszu
...     print(line.upper(), end='')
... StopIteration                                # Wywołuje __next__, przechwytuje
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 32)
```

Warto zwrócić uwagę na parametr `end=' '` funkcji `print()`, który spowoduje, że funkcja ta nie dopisze znaku nowego wiersza `\n` (jak to czyni standardowo) — w przeciwnym razie w wyniku między wierszami otrzymalibyśmy dodatkowy pusty wiersz (w wersji 2.x analogiczną rolę spełnia przecinek dodawany na końcu wywołania funkcji `print`). Metoda ta uznawana jest za *najlepszy* sposób wczytywania plików tekstowych z trzech powodów — jest najprostsza do zapisania w kodzie, najszybsza do wykonania i najlepsza, jeśli chodzi o użycie pamięci. Starszy, oryginalny sposób uzyskania tego samego efektu za pomocą pętli `for` polegał na wywołaniu metody `readlines` pliku do załadowania zawartości pliku do pamięci w postaci listy łańcuchów znaków.

```
>>> for line in open('script2.py').readlines():
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 32)
```

Technika wykorzystująca metodę `readlines()` nadal działa, jednak nie jest uznawana za najlepszą praktykę i gorzej spisuje się w kategorii wykorzystania pamięci. Tak naprawdę, ponieważ ta wersja rzeczywiście ładuje cały plik naraz do pamięci, dla plików większych od rozmiaru pamięci dostępnej na komputerze nie będzie wcale działała. Z kolei wersja oparta na iteratorze nie jest podatna na problemy związane z pamięcią, ponieważ wczytuje jeden wiersz naraz. Co więcej, wersja z iteratorem została w Pythonie mocno zoptymalizowana, dlatego powinna również działać szybciej, choć jest to w dużym stopniu zależne od wersji.

Jak wspominaliśmy w rozdziale 13., w ramce „Warto pamiętać — skanery plików”, zawartość pliku można wczytywać wiersz po wierszu za pomocą pętli `while`.

```
>>> f = open('script2.py')
>>> while True:
```

```

...     line = f.readline()
...     if not line: break
...     print line.upper(),
...
...zawartość pliku...

```



Uwagi do wersji: W Pythonie 2.x metoda iterująca nosi nazwę `X.next()` zamiast `X.__next__()`. W celu zapewnienia możliwości przenoszenia kodu wbudowana funkcja `next(X)` jest również dostępna zarówno w Pythonie 3.x, jak i w 2.x (2.6 i nowszych), i w wersji 3.x wywołuje metodę `X.__next__()`, a w wersji 2.x metodę `X.next()`. Poza tym, że są różnice w nazwach metod, iteracja działa tak samo w 2.x i 3.x. W wersjach 2.6 i 2.7 do ręcznego iterowania powinieneś po prostu używać metody `X.next()` lub funkcji `next(X)` zamiast metody `X.__next__()` z wersji 3.x; w wersjach starszych niż 2.6 zamiast funkcji `next(X)` użyj wywołań metody `X.next()`.

Kod ten będzie jednak najprawdopodobniej działał wolniej od pętli `for` opartej na iteratorze, ponieważ iteratory działają wewnątrz Pythona z szybkością języka C, natomiast wersja z pętlą `while` wykonuje kod bajtowy Pythona za pośrednictwem maszyny wirtualnej Pythona. Za każdym razem gdy wymieniamy kod Pythona na kod języka C, szybkość wzrasta. Ponownie jednak: własność ta nie jest uniwersalna, szczególnie w Pythonie 3.x. Zagadnieniami wydajności zajmiemy się w rozdziale 21., gdzie przedstawimy porównanie efektywności tego typu konstrukcji^[1].

Iterowanie ręczne — `iter` i `next`

Aby uprościć iterowanie ręczne, Python 3.x udostępnia również wbudowaną funkcję `next`, która automatycznie wywołuje metodę `__next__` obiektu. Tak jak wspominaliśmy przed chwilą, takie wywołanie jest również obsługiwane w Pythonie 2.x w celu zapewnienia możliwości przenoszenia kodu. Dla danego obiektu iterowalnego `X` wywołanie `next(X)` jest takie samo jak wywołanie metody `X.__next__()` w wersji 3.x (czy `X.next()` w wersji 2.x), ale jest zauważalnie prostsze i bardziej niezależne od wersji. W przypadku plików możemy stosować dowolną z form:

```

>>> f = open('script2.py')
>>> f.__next__()                               # Bezpośrednie wywołanie metody iteratora
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'
>>> f = open('script2.py')
>>> next(f)                                    # W wersji 3.x wbudowana funkcja nextf)
wywołuje metodę f.__next__()
'import sys\n'
>>> next(f)                                    # next(f) => [3.x: f.__next__()], [2.x:
f.next()]
'print(sys.path)\n'

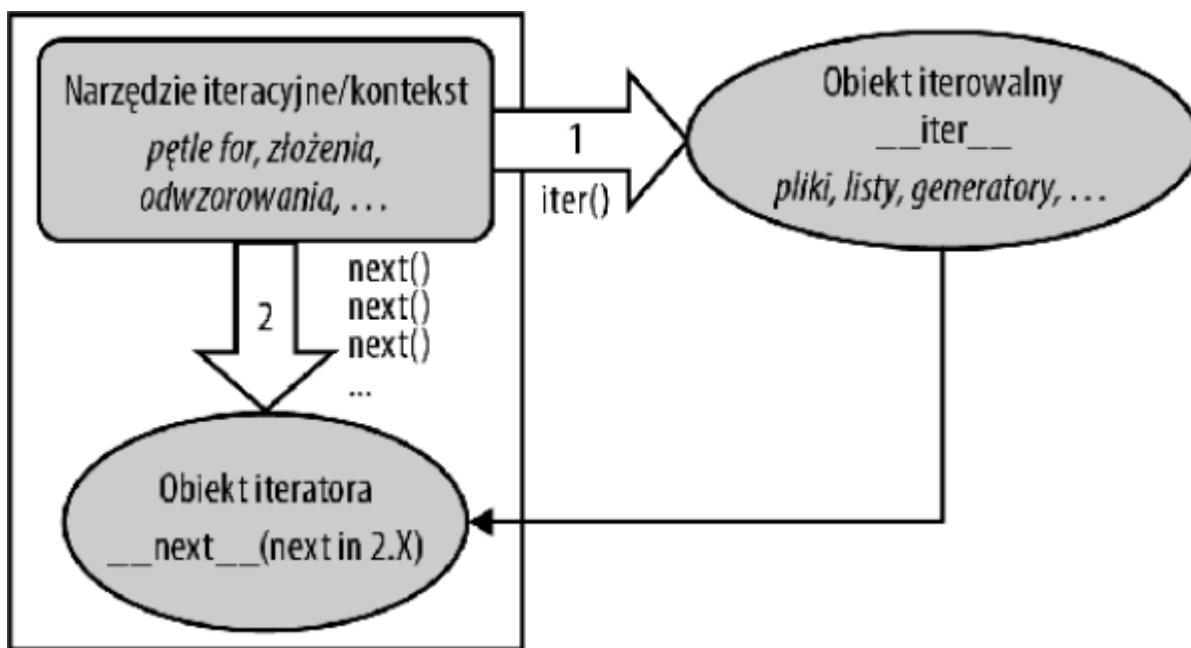
```

Technicznie rzecz biorąc, istnieje jeszcze jeden element protokołu iteratorów, do którego nawiązywaliśmy już wcześniej. Kiedy pętla `for` zaczyna działanie, najpierw pozyskuje iterator z obiektu iterowalnego, przekazując go do wbudowanej funkcji `iter`; obiekt zwracany przez funkcję `iter` posiada wymaganą metodę `next`. Funkcja `iter` wewnętrznie uruchamia metodę `__iter__`, podobnie jak funkcja `next` metodę `__next__`.

Pełny protokół iteracji

Bardziej formalną definicję pokazano na rysunku 14.1, który przedstawia szkic pełnego protokołu iteracji, używanego przez wszystkie narzędzia iteracyjne w Pythonie i obsługiwanej przez wiele różnych typów obiektów. Cały protokół jest tak naprawdę oparty na *dwoch obiektach* używanych w dwóch osobnych krokach przez narzędzia iteracyjne:

- *Obiekt iterowalny* (ang. `iterable`), którego metoda `__iter__` jest uruchamiana przez funkcję `iter`.
- *Obiekt iteratora* (ang. `iterator`) zwracany przez obiekt iterowalny, generujący wartości podczas iteracji, którego metoda `__next__` jest uruchamiana przez funkcję `next` i zgłasza wyjątek `StopIteration` po zakończeniu tworzenia wyników.



Rysunek 14.1. Protokół iteracji w Pythonie, używany w pętlach, listach składanych, odwzorowaniach itd. i obsługiwany przez pliki, listy, słowniki, generatory (omawiane w rozdziale 20.) i inne komponenty. Niektóre obiekty są zarówno kontekstami iteracji, jak i obiektami iterowalnymi, takimi jak wyrażenia generatora i niektóre narzędzia z wersji 3.x (takie jak `map` czy `zip`). Niektóre obiekty są zarówno obiektami iterowalnymi, jak i iteratorami i zwracają same siebie do wywołania funkcji `iter()`, która wtedy nie może działać

W większości przypadków kolejne kroki są automatycznie koordynowane przez narzędzia iteracyjne, ale przedstawiony proces pomaga zrozumieć role tych dwóch obiektów. Na przykład w niektórych przypadkach *ten sam* obiekt jest zarówno obiektem iterowalnym, jak i iteratorem (np. pliki), a *obiekt iteratora* jest często tymczasowy, używany wewnętrznie przez narzędzie iteracji.

Ponadto niektóre obiekty są *zarówno* narzędziem kontekstowym iteracji (iterując), jak i obiektem iterowalnym (ich wyniki są iterowalne) — przykładami mogą być wyrażenia generatora,

omawiane w rozdziale 20., czy `map` i `zip` w Pythonie 3.x. Jak niebawem się przekonasz, kolejne narzędzia stają się iterowalne w wersji 3.x — w tym `map`, `zip`, `range` i niektóre metody słowników — dzięki czemu możemy uniknąć tworzenia w pamięci list wszystkich wyników jednocześnie.

W rzeczywistym kodzie pierwszy krok protokołu iteracji staje się oczywisty, jeżeli przyjrzymy się, jak pętle `for` przetwarzają wewnętrznie wbudowane typy sekwencji, takie jak listy:

```
>>> L = [1, 2, 3]
>>> I = iter(L)                      # Pozyskanie obiektu iteratora z obiektu
iterowalnego
>>> I.__next__()                   # Wywołanie metody next iterator, aby
przejść do kolejnego elementu
1
>>> I.__next__()                   # Zamiast tego w wersji 2.x można użyć
I.next() lub next(I)
2
>>> I.__next__()
3
>>> I.__next__()
...pominieto komunikat o błędzie...
StopIteration
```

Tego typu inicjalizacja nie jest konieczna w przypadku plików, ponieważ obiekt plikowy jest sam w sobie iteratorem. Ponieważ pliki obsługują tylko jedną iterację (nie mogą przechodzić wstecz do obsługi wielu aktywnych skanów), posiadają swoją własną metodę `__next__` i nie muszą zwracać osobnego obiektu:

```
>>> f = open('script2.py')
>>> iter(f) is f
True
>>> iter(f) is f.__iter__()
True
>>> f.__next__()
'import sys\n'
```

Listy i wiele innych wbudowanych obiektów nie są jednak własnymi iteratorami, ponieważ obsługują wiele otwartych, równoległych iteracji — na przykład jednocześnie może istnieć wiele iteracji w zagnieżdżonych pętlach znajdujących się w różnych pozycjach. W przypadku takich obiektów, aby rozpocząć iterację, musimy wywołać funkcję `iter`:

```
>>> L = [1, 2, 3]
>>> iter(L) is L
False
>>> L.__next__()
AttributeError: 'list' object has no attribute '__next__'
```

```
>>> I = iter(L)
>>> I.__next__()
1
>>> next(I)                                # Równoważne I.__next__()
2
```

Iteracje ręczne

Choć narzędzia iteracyjne Pythona wywołują te funkcje w sposób automatyczny, jednak możemy ich użyć w sposób *ręczny*. Poniższy przykład demonstruje równoważność między iteracją automatyczną i ręczną:^[2]

```
>>> L = [1, 2, 3]
>>>
>>> for X in L:                            # Iteracja automatyczna
...     print(X ** 2, end=' ')
...     __next__, przechwytuje wyjątki
...
1 4 9
>>> I = iter(L)                          # Ręczna iteracja: pętla for robi to
samo niejawnie
>>> while True:
...     try:                                # Instrukcja try pozwala obsługiwać
wyjątki
...         X = next(I)                    # Alternatywnie: I.__next__ w wersji 3.x
...     except StopIteration:
...         break
...     print(X ** 2, end=' ')
...
1 4 9
```

Aby zrozumieć ten kod, należy mieć świadomość, że instrukcje `try` służą do wykonywania kodu i przechwytywania wyjątków wywoływanych w tym kodzie (z wyjątkami spotkaliśmy się przez chwilę w rozdziale 11., ale tematykę tę omówimy bardziej szczegółowo w części VII). Warto jeszcze wspomnieć, że pętle `for` i inne formy iteracji w przypadku klas definiowanych przez użytkownika mogą działać inaczej od opisanych tu zasad, na przykład odwołując się do kolejnych indeksów obiektu na liście zamiast stosowania opisanego tu protokołu iteracji. Do tego zagadnienia wróćmy przy okazji omawiania tematu przeciążania operatorów klas w rozdziale 30.

Inne wbudowane typy iterowalne

Poza fizycznymi sekwencjami, takimi jak listy, pozostałe typy również mają przydatne iteratory. Klasycznym sposobem przechodzenia przez kolejne klucze słownika jest na przykład jawne zażądanie listy kluczów:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D.keys():
...     print(key, D[key])
...
a 1
b 2
c 3
```

W nowszych wersjach Pythona słowniki są obiektami iterowalnymi z iteratorem automatycznie zwracającym po jednym kluczu na raz w każdej iteracji:

```
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'b'
>>> next(I)
'c'
>>> next(I)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

W efekcie nie ma już potrzeby używania metody `keys()` do przeglądania kluczy słownika: pętla `for` wykorzysta protokół iteracji do pozyskiwania kluczy po jednym w każdym swoim przebiegu.

```
>>> for key in D:
...     print(key, D[key])
...
a 1
c 3
b 2
```

Bez wnikania w szczegóły warto wspomnieć, że istnieją również inne obiekty Pythona obsługujące protokół iteracyjny, które dzięki temu można wykorzystywać w pętli `for`. Na przykład obiekty typu *shelve* (system plików do zapisu i odczytu obiektów Pythona) oraz wyniki wywołania funkcji `os.popen()` (narzędzie do wywoływania poleceń powłoki z możliwością odczytu zwracanych przez nie wyników, o którym wspominaliśmy w poprzednim rozdziale) również są obiektami iterowalnymi:

```
>>> import os
>>> P = os.popen('dir')
>>> P.__next__()
```

```

\

>>> P.__next__()

' Volume Serial Number is D093-D1F7\n'

>>> next(P)

TypeError: _wrap_close object is not an iterator

```

Zauważ, że w Pythonie 2.x obiekty popen obsługują metodę `P.next()`. W wersji 3.x obsługują metodę `P.__next__()`, ale już nie wbudowaną funkcję `next(P)`. Ponieważ funkcja `next()` wywołuje metodę `__next__`, może to wydawać się niezwykłe, gdyż oba wywołania działają poprawnie, jeżeli użyjemy pełnego protokołu iteracji wykorzystywanego automatycznie przez pętle i inne konteksty iteracji, z jego wywołaniem funkcji `iter` najwyższego poziomu (co powoduje wykonanie wewnętrznych kroków niezbędnych do obsługi kolejnych wywołań funkcji `next` dla tego obiektu):

```

>>> P = os.popen('dir')

>>> I = iter(P)

>>> next(I)

' Volume in drive C has no label.\n'

>>> I.__next__()

' Volume Serial Number is D093-D1F7\n'

```

Również w domenie systemowej standardowy moduł do przetwarzania katalogów z poziomu Pythona, `os.walk`, jest iterowalny w podobny sposób, ale odpowiedni przykład przedstawimy dopiero po omówieniu w rozdziale 20. generatorów i funkcji `yield`, będących podstawami działania tego narzędzia.

Protokół iteracyjny jest również powodem, dla którego musimy opakowywać niektóre z wyników w funkcję `list()`, tak aby wyświetlić wszystkie wartości jednocześnie. Obiekty iterowalne zwracają bowiem po jednym wyniku na raz i nie działają jak lista.

```

>>> R = range(5)

>>> R                               # W 3.x zakresy są obiektami iterowalnymi

range(0, 5)

>>> I = iter(R)                  # Użycie protokołu iteracji

>>> next(I)

0

>>> next(I)

1

>>> list(range(5))            # Użycie listy do jednoczesnej prezentacji
wszystkich wyników

[0, 1, 2, 3, 4]

```

Zauważ, że wywołanie funkcji `list` nie jest wymagane w wersji 2.x (gdzie `range` tworzy prawdziwą listę) i nie jest potrzebne w wersji 3.x dla kontekstów, w których iteracja odbywa się automatycznie (np. w przypadku pętli `for`). Jest ono jednak potrzebne do wyświetlania wartości w wersji 3.x i może być również wymagane, gdy niezbędne będzie zachowanie podobne do listy

lub równoległych iteracji obiektów, które dają wyniki na żądanie w wersji 2.x lub 3.x (więcej na ten temat przedstawimy już niebawem).

Teraz gdy wiemy nieco więcej na temat protokołu iteracyjnego, możemy przeanalizować zachowanie funkcji `enumerate()`, którą poznaliśmy w poprzednim rozdziale.

```
>>> E = enumerate('spam')      # Wynik funkcji enumerate() również jest
obiektem iterowalnym

>>> E
<enumerate object at 0x00000000029B7678>

>>> I = iter(E)

>>> next(I)                  # Wygenerowanie wyników z użyciem protokołu
iteracyjnego

(0, 's')

>>> next(I)                  # Przekształcenie na listę generuje wszystkie
elementy jednocześnie

(1, 'p')

>>> list(enumerate('spam'))
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]
```

Z reguły nie mamy okazji obserwować tych szczegółów w działaniu, ale są one wykorzystywane niejawnie przez mechanizm pętli `for`. W rzeczywistości wszelkie przypadki kolejnego przeglądania elementów w Pythonie wykorzystują protokół iteracyjny. Kolejne przykłady poznamy w następnym punkcie.

Listy składane — wprowadzenie

Poznaliśmy mechanizm działania protokołu iteracyjnego, nadszedł więc czas na poznanie najbardziej powszechnych jego zastosowań. Obok pętli `for` najczęstszą formą użycia protokołu iteracyjnego są listy składane (ang. *list comprehension*).

W poprzednim podrozdziale nauczyliśmy się, jak można za pomocą funkcji `range()` modyfikować listę w miarę przechodzenia przez jej elementy:

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```

Takie rozwiązanie działa, jednak jak wspomnieliśmy wcześniej, może nie być to najbardziej optymalne i najlepsze podejście w Pythonie. Wyrażenia list składanych sprawiają, że tego typu rozwiązania stały się mało efektywne. W poniższym przykładzie pętlę zastąpiliśmy pojedynczym wyrażeniem generującym pożądaną listę wyników.

```
>>> L = [x + 10 for x in L]
```

```
>>> L  
[21, 22, 23, 24, 25]
```

Rezultat jest taki sam, jednak wymaga mniejszej ilości kodu z naszej strony i prawdopodobnie działa znacznie szybciej. Lista składana nie jest dokładnie tym samym co wersja z instrukcją `for`, ponieważ tworzy ona *nowy* obiekt listy (co może mieć znaczenie, jeśli istnieje wiele referencji do oryginalnej listy), dla większości zastosowań jest jednak wystarczająco bliska. Jest to również często stosowane i na tyle wygodne rozwiążanie, że zasługuje na osobne omówienie.

Podstawy list składanych

Z listami składanymi po raz pierwszy spotkaliśmy się w rozdziale 4. Z punktu widzenia składni listy składane pochodzą od konstrukcji w zapisie teorii zbiorów, stosującej operacje do każdego elementu zbioru, jednak do ich używania nie jest potrzebna znajomość całej teorii. W Pythonie dla większości osób listy składane wyglądają po prostu jak pisane od tyłu pętle `for`.

Przyjrzyjmy się dokładniej zaprezentowanemu wyżej przykładowi:

```
>>> L = [x + 10 for x in L]
```

Listy składane zapisywane są w nawiasach kwadratowych, ponieważ są one metodą tworzenia nowej listy. Zaczynają się od dowolnego, zbudowanego przez nas wyrażenia, które wykorzystuje tworzoną przez nas zmienną pętli ($x + 10$). Potem następuje część, którą powinniśmy już rozpoznać jako nagłówek pętli `for`. W części tej wymieniona zostaje zmienna pętli oraz obiekt, po którym będziemy iterować (`for x in L`).

By wykonać wyrażenie, Python wykonuje iterację po `L` wewnętrz interpretera, przypisując `x` do każdego elementu z kolei, i generuje wyniki wykonania lewej strony wyrażenia na przechodzonych elementach. Otrzymany wynik jest nową listą zawierającą $x + 10$ dla każdego `x` z listy `L`.

Z technicznego punktu widzenia listy składane są konstrukcją całkowicie opcjonalną, ponieważ zawsze możemy zbudować listę wyników wyrażenia ręcznie za pomocą pętli `for` dodającej po kolei wyniki do listy w miarę przechodzenia elementów.

```
>>> res = []  
>>> for x in L:  
...     res.append(x + 10)  
...  
>>> res  
[31, 32, 33, 34, 35]
```

Tak naprawdę lista składana wewnętrznie robi dokładnie to samo.

Listy składane mają jednak bardziej zwięzły zapis, a ponieważ ten wzorzec kodu budowania list wyników jest w Pythonie tak często spotykany, w wielu kontekstach okazują się one po prostu bardzo wygodne. Co więcej, w zależności od wersji Pythona i samego kodu programu listy składane mogą działać o wiele szybciej od ręcznie zapisanych instrukcji pętli `for` (często nawet dwukrotnie szybciej), ponieważ ich iteracje wykonywane są wewnętrz interpretera z szybkością języka C, a nie z szybkością kodu Pythona. Z tego powodu, szczególnie w przypadku większych zbiorów danych, używanie ich może być bardzo korzystne z punktu widzenia wydajności.

Wykorzystywanie list składanych w plikach

Przyjrzyjmy się teraz kolejnemu często spotykanemu zastosowaniu list składanych, co pozwoli nam lepiej zrozumieć tę technikę. Jak pamiętamy, obiekt pliku ma metodę `readlines()`, która za jednym razem ładuje plik do listy łańcuchów znaków.

```
>>> f = open('script2.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print (sys.path)\n', 'x = 2\n', 'print (2 ** 32)\n']
```

Takie rozwiązanie działa, jednak wszystkie wiersze wyniku mają na końcu znak nowego wiersza (`\n`). W wielu programach znak ten przeszkadza — musimy na przykład uważać, by przy wyświetlaniu uniknąć podwójnych odstępów. Byłoby miło, gdyby udało nam się w jednym ruchu pozbyć wszystkich znaków nowych wierszy, prawda?

Za każdym razem, gdy rozważamy wykonanie operacji na każdym elemencie sekwencji, znajdujemy się w królestwie list składanych. Zakładając na przykład, że zmienna `lines` z poprzedniego przykładu pozostaje bez zmian, poniższy kod pozwala nam wykonać to zadanie, wywołując na każdym wierszu z listy metodę łańcuchów znaków `rstrip()`, która usuwa białe znaki po prawej stronie. Podobnie zadziałałby również wycinek `line[:-1]`, jednak tylko wtedy, gdybyśmy byli pewni, że wszystkie wiersze zostały poprawnie zakończone znakiem `\n`, a to nie zawsze jest regułą w przypadku ostatniego wiersza w pliku:

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print (sys.path)', 'x = 2', 'print (2 ** 32)']
```

Takie rozwiązanie działa, jednak ponieważ listy składane są kolejnym kontekstem iteracyjnym, podobnie jak proste pętle `for`, nie musimy nawet wstępnie otwierać pliku. Jeżeli otworzymy go w wyrażeniu, lista składana automatycznie wykorzysta omówiony wcześniej protokół iteracji, czyli wczyta po jednym wierszu z pliku na raz, wywołując metodę `next()` obiektu pliku, wykoną na tym wierszu metodę `rstrip()` oraz doda do listy wyników. I znów otrzymujemy to, co chcieliśmy — wynik zastosowania metody `rstrip()` na każdym wierszu pliku.

```
>>> lines = [line.rstrip() for line in open('script2.py')]
>>> lines
['import sys', 'print (sys.path)', 'x = 2', 'print (2 ** 32)']
```

Powysze wyrażenie wiele operacji wykonuje w sposób niejawny, jednak dzięki temu sporo otrzymujemy za darmo — Python przegląda plik wiersz po wierszu i buduje listę wyników operacji w sposób automatyczny. Jest to również wydajny sposób zapisania tej operacji w kodzie — ponieważ większość pracy wykonywana jest wewnątrz interpretera Pythona, rozwiązanie to będzie najprawdopodobniej dużo szybsze od odpowiadającej mu pętli `for`. Co więcej, nie ładuje do pamięci całego pliku na raz, jak robią to inne, alternatywne techniki, co w przypadku większych plików może spowodować znaczący wzrost szybkości działania.

Oprócz swojej wydajności listy składane mają jeszcze jedną zaletę: wykonują sporo pracy w niewielkiej ilości kodu. W naszym przykładzie na każdym wierszu przetwarzanego pliku możemy wykonać dowolną operację znakową. Aby to zilustrować, poniżej przedstawiamy kilka przykładów zastosowania list składanych do wykonywania operacji na plikach, takich jak zamiana wszystkich liter na wielkie i inne, o których mówiliśmy już wcześniej:

```
>>> [line.upper() for line in open('script2.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
>>> [line.rstrip().upper() for line in open('script2.py')]
```

```

['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(X ** 32)']

>>> [line.split() for line in open('script2.py')]

[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(x',
 '**','32')]]

>>> [line.replace(' ', '!') for line in open('script2.py')]

['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(2**!32)\n']

>>> [('sys' in line, line[0]) for line in open('script2.py')]

[(True, 'impor'), (True, 'print'), (False, 'x = 2'), (False, 'print')]

```

Przypomnijmy, że łączenie wywołań metod w łańcuch wywołań w drugim z tych przykładów działa, ponieważ metody łańcuchowe zwracają nowy ciąg, do którego możemy zastosować inną metodę łańcuchową. Ostatni z przykładów pokazuje, w jaki sposób możemy również zbierać wiele wyników, o ile są one zapakowane w kolekcję taką jak krotka lub lista.

	<p>Jedna ważna uwaga: jak wspominaliśmy w rozdziale 9., podczas czyszczenia pamięci obiekty plików są zamykane automatycznie (o ile były nadal otwarte). Z tego samego powodu listy składane również automatycznie zamykają plik, gdy jego tymczasowy obiekt jest czyszczony po wykonaniu wyrażenia. Jednak w dystrybucjach innych niż CPython możesz kodować ręczne zamykanie plików przetwarzanych w pętli, aby zapewnić natychmiastowe zwolnienie wykorzystywanych przez nie zasobów. Więcej informacji na temat zamykania plików znajdziesz w rozdziale 9.</p>
---	--

Rozszerzona składnia list składanych

W rzeczywistości funkcjonalność list składanych w praktyce może być jeszcze bogatsza, a w najpełniejszej formie może nawet stanowić swego rodzaju *minijęzyk iteracji*. Przyjrzyjmy się zatem nieco bliżej ich składni:

Klauzula filtrująca: if

Istnieje jedno szczególnie użyteczne rozszerzenie list składanych — pętla `for` zagnieżdżona w wyrażeniu listy składanej może mieć powiązaną klauzulę `if`, pozwalającą na *odfiltrowanie* tych elementów wyników, dla których test nie jest prawdziwy.

Załóżmy na przykład, że chcemy powtórzyć poprzedni przykład, jednak musimy pobrać jedynie wiersze rozpoczętające się od litery `p` (być może pierwszy znak wiersza jest jakiegoś rodzaju kodem działania). Jest to możliwe dzięki dodaniu do wyrażenia filtrującej klauzuli `if`:

```

>>> lines = [line.rstrip() for line in open('script2.py') if line[0] == 'p']

>>> lines

['print (sys.path)', 'print (2 ** 32)']

```

W powyższym kodzie instrukcja `if` sprawdza każdy wiersz wczytany z pliku, weryfikując, czy jego pierwszym znakiem jest `p`. Jeżeli tak nie jest, wiersz ten jest pomijany w liście wyników. Wyrażenie to jest dość rozbudowane, ale łatwe do zrozumienia, jeżeli przełożymy je na prostą instrukcję pętli `for` (listę składaną można zawsze zamienić na instrukcję `for`, dodając do wyniku elementy iteratora).

```

>>> res = []

>>> for line in open('script2.py'):

```

```

...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print (sys.path)', 'print (2 ** 32)']

```

Takie rozwiązanie z pętlą `for` działa, jednak zamiast jednego wiersza kodu wymaga czterech i prawdopodobnie działa zauważalnie wolniej. W rzeczywistości możesz „wcisnąć” znaczną część logiki interpretacji do samego wyrażenia listy składanej, gdy tego potrzebujesz — wyrażenie przedstawione poniżej działa podobnie do poprzedniego, ale zaznacza tylko wiersze, które kończą się cyfrą (przed znakiem nowego wiersza), filtrując dane za pomocą bardziej wyrafinowanego wyrażenia po prawej stronie (zamień `[-1]` na `[-1:]` dla plików z pustymi wierszami):

```

>>> [line.rstrip() for line in open('script2.py') if line.rstrip()
[-1].isdigit()]
['x = 2']

```

W kolejnym przykładzie filtra pierwsze polecenie `len` podaje całkowitą liczbę wierszy w pliku tekstowym, a drugie usuwa wszystkie białe znaki znajdujące się na początku i końcu wiersza, co pozwala na *pominiecie zliczania pustych wierszy* — jak widać, uzyskujemy taki efekt w jednym wierszu kodu (plik tekstowy użyty w tym przykładzie nie jest dołączony do książki i zawiera opis wszystkich literówek znalezionych w pierwszym szkicu tej książki przez mojego korektora):

```

>>> fname = r'd:\books\5e\lp5e\draft1typos.txt'
>>> len(open(fname).readlines())                                     # Wszystkie
wiersze
263
>>> len([line for line in open(fname) if line.strip() != ''])      # Wiersze
niepuste
185

```

Zagnieżdżone pętle: klauzula `for`

Listy składane mogą się stać nawet bardziej złożone, jeżeli będzie taka konieczność. Mogą na przykład zawierać *zagnieżdżone pętle* zapisane w kodzie jako seria klauzul `for`. Ich pełna składnia pozwala na umieszczenie dowolnej liczby klauzul `for`, a każda z nich może mieć opcjonalną, powiązaną klauzulę `if`.

Poniższy kod tworzy na przykład listę konkatenacji `x + y` dla każdego `x` z jednego łańcucha znaków i każdego `y` z drugiego. W rezultacie tworzymy uporządkowaną listę wszystkich kombinacji znaków z obu łańcuchów:

```

>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

```

Podobnie jak poprzednio, tak i teraz jednym ze sposobów na łatwiejsze zrozumienie tego wyrażenia jest konwersja do postaci polecenia poprzez odpowiednie wcięcie jego części. Poniżej przedstawiono równoważny, ale prawdopodobnie wolniejszy, alternatywny sposób osiągnięcia tego samego efektu:

```

>>> res = []

```

```

>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

```

Pomijając tego typu poziom komplikacji, listy składane często jednak bywają nawet zbyt zwięzłe. Są one przeznaczone dla prostych typów iteracji. W przypadku bardziej złożonych zadań struktura pętli `for` będzie zwykle łatwiejsza do zrozumienia i zmodyfikowania w przyszłości. Jak zawsze w świecie programowania, jeżeli coś staje się dla nas zbyt trudne do zrozumienia, najprawdopodobniej życie tego nie jest najlepszym pomysłem.

Ze względu na ich złożoność zagadnienia związane z listami składanymi najlepiej omawiać stopniowo, małymi kroczkami, zatem na razie zakończymy ten temat. Do list składanych ponownie powrócimy w rozdziale 20. w kontekście narzędzi programowania funkcyjnego i tam zdefiniujemy ich składnię bardziej formalnie, a także przedstawimy dodatkowe przykłady. Jak się sam przekonasz, niebawem okaże się, że listy składane są tak samo związane z *funkcjami*, jak z *instrukcjami* pętli.



Ogólna uwaga dla wszystkich stwierdzeń dotyczących wydajności kodu zawartych w tej książce, list składanych i innych komponentów: względna szybkość działania zależy w dużej mierze od dokładnie przetestowanego kodu i użytej wersji Pythona oraz jest wrażliwa na zmiany wprowadzane w poszczególnych wersjach.

Na przykład w wersjach 2.7 i 3.3 klasycznego CPythona listy składane mogą w niektórych testach być niemal dwukrotnie szybsze niż odpowiadające im pętle `for`, ale tylko nieznacznie szybsze w innych testach, a może nawet nieco wolniejsze w jeszcze innych testach, gdzie zastosowane zostaną klauzule filtrujące `if`.

W rozdziale 21. nauczysz się, jak kodować w Pythonie pomiary czasu, i dowieš się, jak interpretować plik `listcomp-speed.txt`, znajdujący się w pakietie kodów źródłowych do tej książki i zawierający kody przykładów omawianych w tym rozdziale. Na razie powinieneś zapamiętać, że w zakresie testów wydajności uzyskanie absolutnie miarodajnych punktów odniesienia jest również mało prawdopodobne, jak uzyskanie jakiegokolwiek konsensusu w projektach open source!

Inne konteksty iteracyjne

W dalszej części książki przekonamy się, że klasy definiowane przez użytkownika również mogą implementować protokół iteracyjny. Tym bardziej warto znać narzędzia potrafiące korzystać z tego mechanizmu, ponieważ każde narzędzie, które potrafi obsłużyć wbudowane typy iterowalne, będzie potrafiło pracować z dowolnymi iteratorami definiowanymi przez użytkownika.

Dotychczas prezentowałem iteratory w kontekście instrukcji pętli `for`, będącej jednym z podstawowych tematów niniejszego rozdziału. Należy jednak pamiętać, że *każde* wbudowane narzędzie przechodzące przez elementy obiektu od lewej do prawej strony korzysta z protokołu iteracji. Obejmuje to omawiane już wcześniej pętle `for...`

```

>>> for line in open('script2.py'):
...     print(line.upper(), end='')

...
IMPORT SYS
PRINT SYS.PATH
X = 2
PRINT(X ** 32)

```

...ale także wiele innych komponentów. Na przykład listy składane i wbudowana funkcja `map` używają tego samego protokołu iteracji, co pokrewna im pętla `for`. Po zastosowaniu do pliku oba te komponenty wykorzystują iterator obiektu pliku do automatycznego skanowania wiersz po wierszu, pobierając iterator za pomocą metody `__iter__` i wywołując metodę `__next__` w każdej iteracji:

```

>>> uppers = [line.upper() for line in open('script2.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT (SYS.PATH)\n', 'X = 2\n', 'PRINT (X ** 32)\n']
>>> map(str.upper, open('script2.py'))           # map() jest w 3.x obiektem
iterowalnym
<map object at 0x0000000029476D8>
>>> list(map(str.upper, open('script2.py')))
['IMPORT SYS\n', 'PRINT (SYS.PATH)\n', 'X = 2\n', 'PRINT (X ** 32)\n']

```

Z wywołaniem `map` spotkaliśmy się w poprzednim rozdziale (oraz krótko w rozdziale 4.). Jest to wbudowana funkcja, która wywołuje podaną funkcję na każdym elemencie obiektu iterowalnego przekazanego jako argument wywołania. Funkcja `map` jest w swoim działaniu podobna do listy składanej, jednak ma bardziej ograniczony zakres zastosowań, ponieważ wymaga użycia funkcji, a nie dowolnego wyrażenia. W Pythonie 3.x funkcja `map` *zwraca* obiekt iterowalny, zatem w celu wyświetlenia wszystkich wyników naraz musimy przekształcić ten obiekt na listę. Więcej informacji na temat tej zmiany znajdziesz w dalszej części tego rozdziału. Ponieważ listy składane są powiązane z pętlami `for`, powrócimy do nich w rozdziałach 19. i 20.

Wiele innych wbudowanych funkcji Pythona może operować na obiektach iterowalnych. Na przykład `sorted` sortuje elementy obiektów iterowalnych, `zip` łączy elementy z kilku takich obiektów, `enumerate` wylicza elementy wraz z ich indeksem, `filter` zwraca elementy, dla których spełniony jest podany warunek, a `reduce` wykonuje wskazaną funkcję na poszczególnych elementach obiektu iterowalnego. Wszystkie te funkcje pracują na obiektach iterowalnych, a `zip`, `enumerate` i `filter` w Pythonie 3.0 dodatkowo *zwracają* obiekt iterowalny (podobnie jak `map`). Oto przykłady wywołania wymienionych funkcji na otwartym pliku.

```

>>> sorted(open('script2.py'))
['import sys\n', 'print(sys.path)\n', 'print(x ** 32)\n', 'x = 2\n']
>>> list(zip(open('script2.py'), open('script2.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n',
'print(sys.path)\n'),
('x = 2\n', 'x = 2\n'), ('print(x ** 32)\n', 'print(x ** 32)\n')]
>>> list(enumerate(open('script2.py')))

```

```

[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'), (3, 'print(x
** 32)\n')]

>>> list(filter(bool, open('script2.py'))) # nonempty=True
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

>>> import functools, operator
>>> functools.reduce(operator.add, open('script2.py'))

'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

```

Wszystkie te funkcje są narzędziami iteracyjnymi, ale każda ma własne, unikalne zadanie. Funkcje `zip()` i `enumerate()` mieliśmy okazję poznać w poprzednim rozdziale, natomiast `filter()` i `reduce()` wykorzystują programowanie funkcyjne, które będziemy omawiać w rozdziale 19., więc na razie odłożymy ich szczegóły; warto tutaj zauważyć, że używają protokołu iteracji dla plików i innych obiektów iterowalnych.

Funkcję `sorted()`, użytą w powyższym przykładzie, poznaliśmy w rozdziale 4., a w rozdziale 8. wykorzystaliśmy ją w kontekście słowników. Funkcja wbudowana `sorted()` jest odpowiednikiem metody `sort()` obsługiwanej przez sekwencje, ale zwraca nową listę zawierającą posortowane elementy i potrafi przetwarzanie dowolny obiekt iterowalny. Zauważ, że w przeciwieństwie do funkcji `map` i innych, w Pythonie 3.x funkcja `sorted` zwraca rzeczywistą listę zamiast obiektu iterowalnego.

Co ciekawe, protokół iteracji jest dziś jeszcze bardziej wszechobecny w Pythonie, niż mogłyby to sugerować omawiane do tej pory przykłady — w zasadzie *wszystko* we wbudowanym zestawie narzędzi Pythona, które skanują obiekty od lewej do prawej, jest zdefiniowane tak, aby używać protokołu iteracji na tym obiekcie. Obejmuje to nawet takie narzędzia, jak wbudowane funkcje `list` i `tuple` (tworzące z obiektu iterowalnego odpowiednio nową listę lub krotkę) oraz metoda `join` ciągów znaków (która tworzy nowy串, umieszczając wybrany串 znaków między łańcuchami znaków obiektu iterowalnego). Z tego powodu wszystkie z poniższych metod będą działały na otwartym pliku i automatycznie odczytywały po jednym wierszu na raz:

```

>>> list(open('script2.py'))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

>>> tuple(open('script2.py'))
('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n')

>>> '&&'.join(open('script2.py'))
'import sys\n&&print(sys.path)\n&&x = 2\n&&print(x ** 32)\n'

```

Co ciekawe, do tej kategorii należą nawet niektóre narzędzia, których byś o to nie podejrzewał. Na przykład przypisanie sekwencji, test przynależności, przypisanie wycinka i metoda `extend` listy również wykorzystują protokół iteracji do skanowania obiektów, a tym samym automatycznie odczytują plik wiersz po wierszu:

```

>>> a, b, c, d = open('script2.py')                                # Przypisanie
sekwencji

>>> a, d
('import sys\n', 'print(x ** 32)\n')

>>> a, *b = open('script2.py')                                     # Rozszerzona
forma z wersji 3.X

>>> a, b

```

```

('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n'])
>>> 'y = 2\n' in open('script2.py')                                # Test
przynależności
False
>>> 'x = 2\n' in open('script2.py')
True
>>> L = [11, 22, 33, 44]                                         # Przypisanie
wycinka
>>> L[1:3] = open('script2.py')
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n', 44]
>>> L = [11]
>>> L.extend(open('script2.py'))                                     # metoda
list.extend
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

```

Zgodnie z tym, co napisaliśmy w rozdziale 8., funkcja `extend` iteruje automatycznie, ale funkcja `append` już nie — tej drugiej funkcji (lub podobnej) możesz używać, kiedy chcesz dodać obiekt iterowalny do listy bez konieczności wykonywania iteracji, z możliwością iterowania później:

```

>>> L = [11]
>>> L.append(open('script2.py'))                                     # list.append nie
iteruje
>>> L
[11, <_io.TextIOWrapper name='script2.py' mode='r' encoding='cp1252'>]
>>> list(L[1])
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

```

Iteracje są szeroko obsługiwany i bardzo efektywnym rozwiązaniem. We wcześniejszym przykładzie widzieliśmy, że wbudowana funkcja `dict` akceptuje również wynik działania funkcji `zip()` będący obiektem iterowalnym (zobacz rozdziały 8. i 13.). Podobnie działa funkcja `set` oraz *wyrażenia zbiorów i słowników składanych*, dostępne w wersjach 2.7 i 3.x, które mieliśmy okazję poznać w rozdziałach 4., 5. i 8.

```

>>> set(open('script2.py'))
{'print(x ** 32)\n', 'import sys\n', 'print(sys.path)\n', 'x = 2\n'}
>>> {line for line in open('script2.py')}
{'print(x ** 32)\n', 'import sys\n', 'print(sys.path)\n', 'x = 2\n'}
>>> {ix: line for ix, line in enumerate(open('script2.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(x ** 32)\n'}

```

Zbiory i słowniki rozwijane potrafią również wykorzystać rozszerzoną składnię list składanych, którą omówiliśmy wcześniej w tym rozdziale, w tym również warunek `if`:

```
>>> {line for line in open('script2.py') if line[0] == 'p'}
{'print(x ** 32)\n', 'print(sys.path)\n'}
>>> {ix: line for (ix, line) in enumerate(open('script2.py')) if line[0] ==
'p'}
{1: 'print(sys.path)\n', 3: 'print(x ** 32)\n'}
```

Podobnie jak w przykładzie z listą rozwijaną, w tym przypadku również przeglądamy plik wiersz po wierszu, wybieramy z niego wiersze rozpoczynające się literą „p”. W wyniku powstaje zbiór i słownik, ale sporo pracy mamy wykonane za darmo dzięki połączeniu iteracji w pliku oraz składni listy składanej. W dalszej części książki spotkamy kremnego list składanych — wyrażenia generatora — które wdrażają tę samą składnię i działają na obiektach iterowalnych, ale również same w sobie są obiektem iterowalnym:

```
>>> list(line.upper() for line in open('script2.py'))                      # Zobacz
rozdział 20.
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
```

Istnieją również inne funkcje wbudowane obsługujące obiekty iterowalne, ale trudno jest zademonstrować ich działanie na przykładzie plików. Na przykład funkcja `sum` oblicza sumę elementów obiektu iterowalnego; funkcja `any` zwraca wartość `True`, jeżeli dowolny element ma wartość `True`, natomiast funkcja `all` zwraca `True`, jeśli wszystkie elementy mają wartość `True`; a funkcje `max` i `min` zwracają odpowiednio największy i najmniejszy element obiektu iterowalnego. Wszystkie funkcje z przykładu pokazanego poniżej działają na podobnej zasadzie co `reduce`: wykonują operację na wszystkich elementach obiektu iterowalnego, ale zwracają pojedynczy wynik.

```
>>> sum([3, 2, 4, 1, 5, 0])                                              # sum oczekuje samych liczb
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
>>> max([3, 2, 5, 1, 4])
5
>>> min([3, 2, 5, 1, 4])
1
```

Ściśle mówiąc, funkcje `max` i `min` można również zastosować do plików — automatycznie używają one protokołu iteracji do skanowania pliku i wybrania wierszy o odpowiednio najwyższej i najniższej wartości (zastosowanie takiego rozwiązania pozostawię dla Twojej wyobraźni):

```
>>> max(open('script2.py'))                                              # Wiersz z ciągiem znaków o
najwyższej wartości
'x = 2\n'
>>> min(open('script2.py'))
```

```
'import sys\n'
```

Jest jeszcze jeden kontekst iteracji, o którym warto wspomnieć, choć na razie bardziej w formie ciekawostki: w rozdziale 18. zobaczysz, że w wywołaniach funkcji można użyć specjalnej formy `*arg`, aby rozpakować kolekcję wartości do postaci poszczególnych argumentów. Jak możesz się domyślać, taka składnia obsługuje również dowolne obiekty iterowalne, w tym także pliki (więcej szczegółów na temat składni można znaleźć w rozdziale 18.; w rozdziale 20. możesz zapoznać się z sekcją, która rozszerza tę koncepcję na wyrażenia generatora, a w rozdziale 11. znajdziesz wskazówki dotyczące używania funkcji `print` z wersji 3.x w Pythonie 2.x):

```
>>> def f(a, b, c, d): print(a, b, c, d, sep='&')

...
>>> f(1, 2, 3, 4)
1&2&3&4

>>> f(*[1, 2, 3, 4])                                # Rozpakowanie listy argumentów
1&2&3&4

>>>
>>> f(*open('script2.py'))                          # Iteracja po wierszach
import sys
&print(sys.path)

&x = 2
&print(x ** 32)
```

W rzeczywistości, w związku z tym, że składnia rozpakowywania argumentów akceptuje obiekty iterowalne, możemy również wykorzystać funkcję wbudowaną `zip` do *rozpakowywania* spakowanych krotek lub w przypadku zagnieżdżenia wywołań funkcji `zip` (uwaga, analiza poniższego kodu jest niewskazana dla osób o słabych nerwach!).

```
>>> X = (1, 2)
>>> Y = (3, 4)
>>>
>>> list(zip(X, Y))                                # pakowanie krotek: zwraca obiekt
iterowalny
[(1, 3), (2, 4)]

>>>
>>> A, B = zip(*zip(X, Y))                         # Rozpakowanie zipa!
>>> A
(1, 2)
>>> B
(3, 4)
```

Istnieją również inne narzędzia Pythona zwracające obiekty iterowalne zamiast gotowych list wyników. Są to między innymi funkcja `range()` oraz obiekty widoku słownika. W następnym podrozdziale omówimy szczegóły nowości w protokole iteracyjnym wprowadzone w Pythonie 3.x.

Nowe obiekty iterowalne w Pythonie 3.x

Jednym z podstawowych wyróżników Pythona 3.x jest to, że w porównaniu z wersjami 2.x kładzie silniejszy nacisk na iteratory. To, wraz z modelem Unicode i obowiązkowymi klasami w nowym stylu, jest jedną z najbardziej rozległych zmian w wersji 3.x.

Obok iteratorów związanych z typami wbudowanymi, jak pliki czy słowniki, metody słowników `keys`, `values` i `items` również zwracają obiekty iterowalne. Podobna zmiana spotkała funkcje wbudowane `range`, `map`, `zip` i `filter`. Jak mieliśmy okazję przekonać się w poprzednim rozdziale, ostatnie trzy z nich przetwarzają obiekty iterowalne, jak również je zwracają. Konstrukcje te w Pythonie 3.x generują wyniki na żądanie, zamiast zwracać gotowe listy, jak ma to miejsce w Pythonie 2.x.

Wpływ na kod w wersji 2.x — zalety i wady

Takie podejście oszczędza pamięć, ale w niektórych kontekstach może mieć wpływ na styl programowania. Na przykład w różnych miejscach tej książki musielibyśmy opakowywać niektóre wyniki wywołań funkcji i metod w wywołanie funkcji `list(...)`, aby zmusić je do *wyswietlania* wszystkich wyników naraz:

```
>>> zip('abc', 'xyz')                                # Obiekt iterowalny w 3.x, lista
w 2.x
<zip object at 0x000000000294C308>
>>> list(zip('abc', 'xyz'))                         # Wymuszenie wyświetlenia listy
wyników w 3.x
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Podobna konwersja jest wymagana, jeżeli chcemy zastosować operacje listy lub *sekwencji* do większości obiektów iterowalnych, generujących elementy na żądanie — na przykład do indeksowania, dzielenia lub łączenia obiektów iterowalnych. Wyniki działania takich narzędzi w wersji 2.x obsługują takie operacje bezpośrednio:

```
>>> Z = zip((1, 2), (3, 4))      # W przeciwieństwie do list zwracanych w
wersji 2.x nie możemy tutaj
                                         # np. indeksować
>>> Z[0]
TypeError: 'zip' object is not subscriptable
```

Jak się przekonasz w rozdziale 20., konwersja na listy może być bardziej subtelnie wymagana do obsługi *wielu równoległych iteracji* dla nowych narzędzi iteracyjnych, które obsługują tylko jedno skanowanie, takich jak funkcje `map` czy `zip` — w przeciwieństwie do „listowych” postaci z wersji 2.x, ich wartości w wersji 3.x są wyczerpane po jednej iteracji:

```
>>> M = map(lambda x: 2 ** x, range(3))
>>> for i in M: print(i)
...
1
2
4
```

```

>>> for i in M: print(i)          # W przeciwnieństwie do list z 2.x mamy tutaj
      tylko jedną iterację

                                         # (tak samo z

funkcją zip)

...
>>>

```

Taka konwersja w Pythonie 2.x nie jest potrzebna, ponieważ takie funkcje jak `zip()` zwracają gotową listę wyników. W wersji 3.x zwracają jednak obiekt iterowalny, który generuje kolejne wyniki na żądanie. Może to powodować problemy z poprawnym działaniem kodu z wersji 2.x i oznacza, że w przypadku konieczności pobrania wszystkich wyników naraz, na przykład w celu wyświetlenia ich w konsoli, musimy dokonać dodatkowego przekształcenia na listę. Przyjrzyjmy się niektórym nowościom Pythona 3.x związanym z obiektami iterowalnymi.

Obiekt iterowalny range

Podstawowe zastosowanie funkcji `range()` omówiliśmy w poprzednim rozdziale. W Pythonie 3.x funkcja `range()` zwraca obiekt iterowalny, który na żądanie generuje liczby z zadanego zakresu bez zużywania pamięci na całą listę wyników. Funkcja ta jest odpowiednikiem i następcą funkcji `xrange()` z Pythona 2.x (patrz uwaga na temat zgodności). Aby uzyskać listę kolejnych wartości z zadanego zakresu (na przykład w celu ich wypisania w konsoli), należy zastosować `list(range(...))`:

```

C:\code> c:\python33\python

>>> R = range(10)                  # range() zwraca obiekt
iterowalny, a nie listę

>>> R
range(0, 10)

>>> I = iter(R)                  # Przekształcenie obiektu
iterowalnego range na iterator

>>> next(I)                     # Przejście do następnej wartości
                                # Pętle for, rozwinięcia itp.

0
wykonują to w sposób niejawny

>>> next(I)
1

>>> next(I)
2

>>> list(range(10))              # Przekształcenie na listę,
jeżeli to konieczne
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

W przeciwnieństwie do list zwracanych w Pythonie 2.x obiekt `zakresu` w Pythonie 3.x obsługuje tylko iterację, indeksowanie i funkcję `len()`. Nie obsługuje pozostałych operacji typowych dla sekwencji (aby je wykorzystać, należy przekształcić zakres na listę z użyciem funkcji `list(...)`).

```

>>> len(R)                        # range obsługuje iteracje, funkcję
len() oraz indeksowanie, ale nic więcej

```

```

10
>>> R[0]
0
>>> R[-1]
9
>>> next(I)                                # Kontynuacja iteratora od ostatniego
miejsca
3
>>> I.__next__()                           # .next() działa tak samo jak
__next__(), ale zaleca się stosować next()
4

```



Uwaga na temat zgodności wersji: Jak wspominaliśmy już w poprzednim rozdziale, Python 2.x posiada wbudowaną funkcję `xrange()`, która działa analogicznie do `range()`, ale zwraca obiekt generujący kolejne wartości na żądanie, zamiast tworzyć od razu całą listę w pamięci. To dokładnie to samo, co w Pythonie 3.x robi funkcja `range()`, dlatego funkcja `xrange()` nie jest już dostępna w wersji 3.x. W programach dla wersji 2.x nadal możesz ją spotkać i jej używać, szczególnie ze względu na fakt, że jej odpowiednik `range()` jest mało efektywny pod względem użycia pamięci.

Jak wyjaśnialiśmy w poprzednim rozdziale, metoda `file.readlines()` stosowana w celu zminimalizowania zużycia pamięci w wersji 2.x w Pythonie 3.x została z podobnych powodów porzucona na korzyść iteratorów plików.

Obiekty iterowalne `map`, `zip` i `filter`

Funkcje wbudowane `map()`, `zip()` i `filter()` w Pythonie 3.x stały się obiektami iterowalnymi, podobnie jak funkcja `range()`. Dzięki temu swoje wyniki generują na bieżąco, bez konieczności tworzenia i umieszczania w pamięci całej listy wyników naraz. Wszystkie trzy wymienione funkcje nie tylko mogą przetwarzarć obiekty iterowalne, jak to było w wersji 2.x, ale dodatkowo w wersji 3.x zwracają wyniki w postaci iterowalnej. Jednak w przeciwieństwie do funkcji `range()` funkcje te są swoimi własnymi iteratorami, które „zużywają” swoje wyniki w trakcie przetwarzania. Innymi słowy, na takich wynikach nie można uruchomić kilku niezależnych iteratorów przetwarzających je równolegle w różnych miejscach iteracji.

Poniższy listing przedstawia przykład użycia funkcji wbudowanej `map()`, którą przedstawiliśmy w poprzednim rozdziale. Podobnie jak w przypadku innych obiektów iterowalnych, w razie potrzeby wyniki funkcji `map()` możemy przekształcić na listę za pomocą wywołania funkcji `list(...)`, ale domyślna mechanika iteratorów pozwala w przypadku dużych porcji danych zaoszczędzić sporą ilość pamięci.

```

>>> M = map(abs, (-1, 0, 1))                  # map() zwraca obiekt iterowalny, a
nie listę
>>> M
<map object at 0x00000000029B75C0>
>>> next(M)                                    # Ręczne użycie iteratora, „zużywa”
wyniki

```

```

1                                         # Iterator nie obsługuje funkcji
len() ani indeksowania

>>> next(M)
0

>>> next(M)
1

>>> next(M)
StopIteration

>>> for x in M: print(x)             # Iterator zwrócony przez funkcję
map() jest już pusty po jednym przejściu
...
>>> M = map(abs, (-1, 0, 1))        # Ponowna próba wymaga utworzenia
nowego obiektu iterowalnego oraz iteratora
>>> for x in M: print(x)           # Konteksty iteracyjne automatycznie
używają funkcji next()
...
1
0
1
>>> list(map(abs, (-1, 0, 1)))     # Iterator można przekształcić w
listę
[1, 0, 1]

```

Funkcja wbudowana `zip()` przedstawiona w poprzednim rozdziale sama w sobie jest kontekstem iteracyjnym, ale zwraca również obiekt iterowalny z iteratorem działającym na tych samych zasadach:

```

>>> Z = zip((1, 2, 3), (10, 20, 30))    # zip() działa tak samo: zwraca
jednorazowy iterator

>>> Z
<zip object at 0x0000000002951108>

>>> list(Z)
[(1, 10), (2, 20), (3, 30)]

>>> for pair in Z: print(pair)          # Po pełnym przebiegu elementy są
wyczerpane...
...
>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> for pair in Z: print(pair)          # Iterator może być użyty ręcznie
lub automatycznie
...

```

```

(1, 10)
(2, 20)
(3, 30)

>>> Z = zip((1, 2, 3), (10, 20, 30))      # Ręczna iteracja (wywołanie iter())
nie jest potrzebne)

>>> next(Z)
(1, 10)

>>> next(Z)
(2, 20)

```

Zbliżone właściwości ma wynik działania funkcji wbudowanej `filter()`, którą spotkaliśmy już w rozdziale 12. i której bardziej szczegółowo przyjrzymy się w dalszej części książki. Funkcja zwraca wyniki w obiekcie iterowalnym, dla którego przekazana funkcja ma wartość True (jak pamiętasz, w Pythonie wartość True mają niepuste obiekty, a funkcja `bool` zwraca wartość „prawdy” obiektu):

```

>>> filter(bool, ['spam', '', 'ni'])
<filter object at 0x00000000029B7B70>
>>> list(filter(bool, ['spam', '', 'ni']))
['spam', 'ni']

```

Jak większość narzędzi omawianych w tym podrozdziale, funkcja `filter()` w Pythonie 3.x może przetwarzać obiekty iterowalne i zwraca inne obiekty iterowalne, zawierające wyniki działania. Funkcję `filter` można również emulować za pomocą rozszerzonej składni listy składanej, która automatycznie testuje wartości prawdy:

```

>>> [x for x in ['spam', '', 'ni'] if bool(x)]
['spam', 'ni']
>>> [x for x in ['spam', '', 'ni'] if x]
['spam', 'ni']

```

Iteratory wielokrotne kontra pojedyncze

Warto zwrócić uwagę na różnice między wynikiem funkcji wbudowanej `range()` a wynikami innych funkcji wbudowanych omówionych w niniejszym rozdziale. Wynik ten obsługuje sprawdzanie długości z użyciem funkcji `len()` i indeksowanie i nie jest iteratorem (to znaczy w kontekście iteracyjnym iterator jest tworzony przez wywołanie funkcji `iter()`). Co najciekawsze, obiekt ten obsługuje wielokrotną równoległą iterację elementów.

```

>>> R = range(3)                                # range() pozwala na wielokrotną
iterację

>>> next(R)
TypeError: range object is not an iterator

>>> I1 = iter(R)

>>> next(I1)
0

```

```

>>> next(I1)
1
>>> I2 = iter(R)                                # Dwa iteratory jednego obiektu
>>> next(I2)
0
>>> next(I1)                                    # I1 znajduje się na innej
pozycji niż I2
2

```

Dla porównania: w wersji 3.x funkcje `zip`, `map` i `filter` nie obsługują wielu aktywnych iteratorów dla tego samego wyniku; z tego powodu wywołanie funkcji `iter` jest opcjonalne dla przeglądania wyników takich obiektów — same dla siebie są iteratorami (w wersji 2.x funkcje te zwracają listy wielokrotnego skanowania, więc dla nich te uwagi nie mają zastosowania):

```

>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z)                                # Dwa iteratory jednego wyniku
funkcji zip()
>>> next(I1)
(1, 10)
>>> next(I1)
(2, 11)
>>> next(I2)                                    # (3.x) I2 znajduje się na tej
samej pozycji co I1!
(3, 12)
>>> M = map(abs, (-1, 0, 1))                  # Tak samo działa wynik map()
oraz filter()
>>> I1 = iter(M); I2 = iter(M)
>>> print(next(I1), next(I1), next(I1))
1 0 1
>>> next(I2)                                    # (3.x) wyniki wyczerpane po
jednym skanowaniu
StopIteration
>>> R = range(3)                                # Natomiast wynik range() pozwala
na wielokrotną iterację
>>> I1, I2 = iter(R), iter(R)
>>> [next(I1), next(I1), next(I1)]
[0 1 2]
>>> next(I2)                                    # Wiele aktywnych skanów,
podobnie jak z listami w 2.x

```

W rozdziale 30. zajmiemy się tworzeniem własnych klas iterowalnych i zobaczymy, że obiekty obsługujące wielokrotną iterację w wyniku działania funkcji `iter()` zwracają z reguły nowy, specjalizowany obiekt iteratora, natomiast obiekty nieobsługujące wielokrotnej iteracji same są iteratorami, co zwykle oznacza, że w wyniku działania funkcji `iter()` zwracają same siebie. W rozdziale 20. przekonasz się, że *funkcje i wyrażenia generatorów* zachowują się w sposób zbliżony do wyniku funkcji `zip()`, obsługując pojedynczą iterację. W tym samym rozdziale będziemy mieli okazję przeanalizować subtelne konsekwencje prób wykorzystania obiektów nieobsługujących wielokrotnej iteracji w pętlach, w których usiłuje się przeglądać je wiele razy — kod, który uprzednio traktował takie wyniki jak listę, w nowych wersjach Pythona może nie działać poprawnie bez ręcznego przeprowadzenia konwersji wyników do postaci listy.

Obiekty iterowalne — widoki słownika

W rozdziale 8. wspomniałem o tym, że w Pythonie 3.x metody słownikowe `keys()`, `values()` i `items()` zwracają iterowalne obiekty *widoku* słownika, generujące po jednym wyniku na raz, a nie gotowe listy wartości, jak to miało miejsce w poprzednich wersjach tego języka. Widoki są również dostępne w wersji 2.7 jako opcja, ale kryją się pod specjalnymi nazwami metod, aby uniknąć negatywnego wpływu na istniejący kod. Obiekty widoku słownika zachowują kolejność elementów słownika i odzwierciedlają wprowadzane zmiany. Wzbogaceni o wiedzę dotyczącej obiektów iterowalnych, możemy poznać pozostałe szczegóły tej historii — w Pythonie 3.3 wygląda to następująco (w Twoim systemie kolejność wyświetlanych kluczy może być inna):

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> K = D.keys()                                     # W 3.x obiekt widoku, nie lista
>>> K
dict_keys(['a', 'b', 'c'])
>>> next(K)                                         # Widoki nie są iteratorami
TypeError: dict_keys object is not an iterator
>>> I = iter(K)                                      # Widoki słowników mają
iteratory,
>>> next(I)                                         # które można obsługiwać ręcznie,
' a'                                                 # ale nie obsługują len() i
indeksowania
>>> next(I)
'b'
>>> for k in D.keys(): print(k, end=' ')      # W kontekstach iteracyjnych
odbywa się to automatycznie
...
a b c
```

Tak jak w przypadku wszystkich obiektów iterowalnych, które generują wartości na żądanie, w wersji 3.x zawsze możesz zmusić widok słownika do zbudowania prawdziwej listy, przekazując ją do wbudowanej funkcji `list`. Z reguły nie jest to konieczne, chyba że w celu wyświetlenia na

ekranie zawartości widoku w trybie interaktywnym lub w celu wykonania operacji typowych dla list, takich jak indeksowanie:

```
>>> K = D.keys()
>>> list(K)                                     # Przekształcenie w listę
['a', 'b', 'c']

>>> V = D.values()                             # Podobnie dla wyniku metod
values() i items()

>>> V
dict_values([1, 2, 3])

>>> list(V)                                    # Potrzebujemy list() do
wyświetlania bądź indeksowania jako listy
[1, 2, 3]

>>> V[0]
TypeError: 'dict_values' object does not support indexing

>>> list(V)[0]
1

>>> list(D.items())
[('a', 1), ('b', 2), ('c', 3)]

>>> for (k, v) in D.items(): print(k, v, end=' ')
...
a 1 b 2 c 3
```

Dodatkowo w Pythonie 3.x słowniki są obiektami iterowalnymi, które mają własne iteratory zwracające kolejne klucze, dzięki czemu w takim kontekście nie ma konieczności bezpośredniego używania metody `keys()`:

```
>>> D = {'a': 1, 'b': 2, 'c': 3} # Słowniki oferują własny iterator

>>> I = iter(D) # w każdej iteracji zwracający kolejny klucz

>>> next(I)
'a'

>>> next(I)
'b'

>>> for key in D: print(key, end=' ') # Metoda keys() nie jest niezbędna do iteracji po kluczu,
niezbędna do iteracji po kluczu,
...
# ale w 3.x keys() również zwraca obiekt iterowalny!

a b c
```

Pamiętaj, że metoda `keys()` nie zwraca już listy kluczy, więc tradycyjnie stosowany w starszych wersjach sposób przeglądania słownika po posortowanej liście kluczy nie zadziała w wersji 3.x. W takim przypadku powinieneś w pierwszej kolejności przekształcić wynik metody `keys()` na listę i wykonać funkcję `sorted()` na widoku kluczy bądź samym słowniku, tak jak to zostało pokazane poniżej. Widzieliśmy już to co prawda w rozdziale 8., ale jest to na tyle ważne dla programistów wersji 2.x migrujących swój kod do wersji 3.x, że pokażemy to jeszcze raz:

```
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> for k in sorted(D.keys()): print(k, D[k], end=' ')
...
a 1 b 2 c 3
>>> for k in sorted(D): print(k, D[k], end=' ')    # Zalecana technika
sortowania kluczy
...
a 1 b 2 c 3
```

Inne zagadnienia związane z iteracjami

Jak już wielokrotnie wspominaliśmy, w rozdziale 20. omówimy więcej zagadnień związanych z listami składanymi i obiektami iterowalnymi, tym razem w powiązaniu z funkcjami, natomiast w rozdziale 30. wróćmy do tych zagadnień w kontekście klas. Między innymi omówimy tam następujące zagadnienia:

- funkcje zdefiniowane przez użytkownika mogą być przekształcone w *generatory obiektów iterowalnych* — służy do tego instrukcja `yield`,
- listy składane ujęte w nawiasy okrągłe (zamiast kwadratowych) stają się *wyrażeniami generatorów* zwracających obiekty iterowalne,
- klasy zdefiniowane przez użytkownika stają się iterowalne dzięki *przeciążaniu operatora `__iter__` lub `__getitem__`*.

W szczególności obiekty iterowalne zdefiniowane przez użytkownika za pomocą klas pozwalają na użycie dowolnych obiektów i operacji w dowolnym kontekście iteracyjnym, który poznaliśmy w tym rozdziale. Obsługując tylko jedną operację — iterację — obiekty takie mogą być używane w wielu różnych kontekstach i narzędziach.

Podsumowanie rozdziału

W niniejszym rozdziale zapoznaliśmy się z instrukcjami pętli Pythona. Po raz pierwszy przyjrzaliśmy się także listom składanym oraz *protokołowi iteracji* Pythona — sposobowi, dzięki któremu obiekty niebędące sekwencjami mogą być przetwarzane sekwencyjnie w pętlach. Jak widzieliśmy, listy składane stosujące wyrażenia do wszystkich elementów obiektu, po którym można iterować, przypominają pętle `for`. Przyjrzaliśmy się różnym funkcjom wbudowanym obsługującym protokół iteracji, jak również przeanalizowaliśmy zmiany w tym zakresie wprowadzone w Pythonie 3.x.

To kończy nasze omówienie poszczególnych instrukcji proceduralnych. Następny rozdział zamknie tę część książki, omawiając opcje dokumentacji kodu Pythona. Dokumentacja, choć nieco odbiegająca od bardziej szczegółowych aspektów kodowania, jest również częścią ogólnego modelu składni i jest ważnym składnikiem dobrze napisanych programów. W kolejnym rozdziale, poprzedzającym zagadnienia związane z większymi strukturami, takimi jak funkcje, zajmiemy się również zbiorem ćwiczeń dotyczących tej części książki. Jak zawsze jednak warto najpierw za pomocą quizu przećwiczyć zagadnienia przedstawione w tym rozdziale.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób powiązane są pętle `for` i obiekty iterowalne?
2. W jaki sposób powiązane są pętle `for` i listy składane?
3. Podaj cztery konteksty iteracyjne w języku Python.
4. Jaki jest obecnie najlepszy sposób odczytywania danych wiersz po wierszu z pliku tekstowego?
5. Używanie jakiego rodzaju broni lub środków nacisku spodziewałbyś się po hiszpańskiej inkwizycji?

Sprawdź swoją wiedzę — odpowiedzi

1. Pętla `for` używa *protokołu iteracji* do przechodzenia między elementami w obiekcie iterowalnym, przez który iteruje. Najpierw pobiera iterator z obiektu iterowalnego, przekazując ten obiekt do funkcji `iter`, a następnie (w wersji 3.x) w każdej iteracji wywołuje metodę `__next__` obiektu iteratora i przechwytuje wyjątek `StopIteration`, aby określić, kiedy przerwać działanie pętli. W wersji 2.x metoda nosi nazwę `next` i jest uruchamiana przez wbudowaną funkcję `next`, dostępną zarówno w 3.x, jak i 2.x. Każdy obiekt obsługujący ten model działa w pętli `for` i we wszystkich innych kontekstach iteracji. W przypadku niektórych obiektów, które są swoimi własnymi iteratorami, początkowe wywołanie funkcji `iter` jest wprawdzie niepotrzebne, ale zupełnie nieszkodliwe.
2. Obie są narzędziami i kontekstami iteracyjnymi. Listy składane są zwięzły i wydajnym sposobem realizacji często spotykanego zastosowania pętli `for` — zbierania wyników zastosowania wyrażenia do wszystkich elementów przechodzonego obiektu. Listę składaną zawsze można przełożyć na pętlę `for`, a część wyrażenia listy składanej wygląda nawet jak nagłówek pętli `for`.
3. Konteksty iteracyjne w Pythonie obejmują pętlę `for`, listy składane, wbudowaną funkcję `map()`, wyrażenie sprawdzające przynależność `in`, a także wbudowane funkcje `sorted()`, `sum()`, `any()` oraz `all()`. Do kategorii tej zaliczamy również wbudowane funkcje `list()` i `tuple()`, metodę łańcuchów znaków `join` i przypisania sekwencji — wszystkie wykorzystują protokół iteracji (zobacz odpowiedź na pytanie 1.) do przechodzenia przez obiekty iterowalne po jednym elemencie na raz.
4. Najlepszą metodą wczytania wierszy pliku tekstowego wcale nie jest obecnie jawne ich załadowanie. Zamiast tego należy otworzyć plik wewnątrz kontekstu iteracyjnego, takiego jak pętla `for` lub lista składana, i pozwolić narzędziu iteracyjnemu na automatyczne odczytywanie wiersz po wierszu dzięki wywoływaniu

metody `next()` pliku w każdej iteracji. Takie podejście jest najlepsze z punktu widzenia prostoty kodu, szybkości wykonania i wymagań związanych z miejscem w pamięci.

5. Każda z następujących odpowiedzi będzie uważana za poprawną: strach, zastraszanie, piękne czerwone stroje, wygodna kanapa i miękkie poduszki.

[1] Uwaga: iterator plików nadal wydaje się być nieco szybszy niż metoda `readlines` i co najmniej 30% szybszy niż pętla `while` zarówno w wersji 2.7, jak i 3.3, w testach, które uruchamiałem z przykładami z tego rozdziału na pliku 1000-wierszowym (pętla `while` w wersji 2.7 była dwukrotnie wolniejsza). Oczywiście obowiązują tutaj wszystkie standardowe zastrzeżenia dotyczące testów porównawczych — otrzymane wyniki są miarodajne tylko dla używanych przeze mnie wersji Pythona, mojego komputera i mojego pliku testowego, a użycie Pythona 3.x dodatkowo komplikuje jeszcze takie analizy, ponieważ jego biblioteki wejścia/wyjścia zostały napisane niemal od nowa tak, aby obsługiwały teksty Unicode i były mniej zależne od systemu. W rozdziale 21. omówimy narzędzia i techniki, których można użyć do samodzielnego pomiaru wydajności takich pętli.

[2] Technicznie rzecz biorąc, pętla `for` wywołuje wewnętrzny odpowiednik metody `I.__next__`, zamiast użytej tutaj funkcji `next(I)`, chociaż rzadko bywa między nimi jakakolwiek różnica. W ręcznych iteracjach można zasadniczo użyć dowolnego schematu wywołania.

Rozdział 15. Wprowadzenie do dokumentacji

Ta część książki kończy się przedstawieniem technik i narzędzi wykorzystywanych w dokumentowaniu kodu Pythona. Choć kod Pythona został zaprojektowany tak, by sam z siebie był czytelny, kilka dobrze rozmieszczonych, zrozumiałych dla użytkowników komentarzy może pomóc innym osobom pojąć sposób działania naszego programu. Jak się przekonasz, Python posiada zarówno odpowiednie polecenia, jak i narzędzia ułatwiające tworzenie dokumentacji kodu. W szczególności omawiany w tym rozdziale system *PyDoc* może tworzyć wewnętrzną dokumentację modułu jako zwykły tekst wyświetlany w powłoce lub jako dokumenty HTML w przeglądarce internetowej.

Choć jest to koncepcja związana raczej z narzędziami, prezentuję ją w tym miejscu książki częściowo dlatego, że dotyczy ona modelu składni Pythona, a po części z myślą o osobach, które próbują zrozumieć zbiór narzędzi Pythona. Z tego drugiego powodu rozszerzymy tutaj nieco wskazówki dotyczące dokumentacji, podane jeszcze w rozdziale 4. Jak zwykle, ponieważ rozdział ten zamyka kolejną część książki, oprócz quizu podsumowującego kończy się ostrzeżeniami dotyczącymi często spotykanych pułapek, na które możesz się natknąć podczas tworzenia swoich programów, a także zbiorem ćwiczeń podsumowujących tę część książki.

Źródła dokumentacji Pythona

W tym miejscu książki większość osób powinna już zaczynać rozumieć, że Python zawiera niesamowitą ilość gotowych możliwości i opcji — wbudowanych funkcji i wyjątków, zdefiniowanych atrybutów i metod obiektów, modułów biblioteki standardowej. Co więcej, tak naprawdę jedynie powierzchownie przyjrzałem się każdej z tych kategorii.

Jednym z pierwszych pytań zadawanych przez początkujących użytkowników Pythona jest często: „Gdzie mogę znaleźć informacje na temat tych wszystkich wbudowanych narzędzi?”. Niniejszy podrozdział zawiera wskazówki dotyczące różnych źródeł dokumentacji dostępnych w Pythonie i prezentuje również *notki dokumentacyjne* (*notki docstrings*) oraz system *PyDoc*, który z nich korzysta. Te zagadnienia pozostają może na peryferiach samego języka, jednak stają się niezbędną wiedzą, kiedy nasz kod osiąga poziom przykładów i ćwiczeń z tej części książki.

W tabeli 15.1 przedstawiono różne miejsca, w których można szukać informacji dotyczących Pythona, zgodnie ze wzrastającym poziomem szczegółowości. Ponieważ dokumentacja jest tak istotnym narzędziem praktycznego programowania, każdą z tych kategorii omówimy za chwilę z osobna.

Tabela 15.1. Źródła dokumentacji Pythona

Forma	Rola
Komentarze ze znakiem #	Dokumentacja w pliku
Funkcja dir	Lista atrybutów dostępnych w obiektach

Notki dokumentacyjne — <code>__doc__</code>	Dokumentacja w pliku dołączana do obiektów
PyDoc — funkcja <code>help</code>	Interaktywna pomoc dla obiektów
PyDoc — raporty HTML	Dokumentacja modułów w przeglądarce
Narzędzie Sphinx	Bogatsza dokumentacja dla większych projektów
Zbiór standardowej dokumentacji	Oficjalne opisy języka i biblioteki
Zasoby internetowe	Samouczki, przykłady, artykuły dostępne w internecie
Publikowane książki	Komercyjne teksty i podręczniki

Komentarze ze znakami

Jak już się mogłeś przekonać, komentarze ze znakami `#` są najbardziej podstawowym sposobem dokumentowania kodu. Python po prostu ignoruje tekst znajdujący się za znakiem `#` (o ile nie znajduje się on wewnątrz literała łańcucha znaków), dlatego można po nim umieścić słowa i opisy znaczące dla programistów. Takie komentarze dostępne są jednak jedynie w plikach źródłowych. Aby zamieścić w kodzie komentarze dostępne nieco szerzej, trzeba będzie skorzystać z notek dokumentacyjnych *docstrings*.

Zgodnie z najlepszymi praktykami programowania notki *docstrings* najlepiej nadają się do tworzenia większej dokumentacji funkcjonalnej (typu „mój plik robi...”), a komentarzy ze znakami `#` lepiej jest używać do dokumentowania mniejszych części kodu (na przykład „to dziwne wyrażenie wykonuje...”), a w szczególności do opisywania pojedynczych poleceń lub niewielkich grup poleceń w kodzie programu. Więcej szczegółowych informacji na temat notek *docstrings* przedstawimy już za chwilę, ale najpierw opowiemy o tym, jak eksplorować obiekty.

Funkcja dir

Jak już pokazywaliśmy wcześniej, wbudowana funkcja `dir` to łatwa metoda na pobranie listy wszystkich atrybutów dostępnych wewnątrz obiektu (na przykład jego metod i prostszych elementów danych). Można ją wywoływać bez żadnych argumentów, aby wyświetlić zmienne dostępne w kontekście wywołania. Co bardziej użyteczne, może być również wywoływana na każdym obiekcie, który posiada atrybuty, w tym zaimportowane moduły i typy wbudowane, a także nazwę typu danych. Na przykład, aby dowiedzieć się, co jest dostępne w wybranym *module*, takim jak moduł `sys` biblioteki standardowej, powinieneś go zaimportować i następnie przekazać jego nazwę do funkcji `dir`:

```
>>> import sys
>>> dir(sys)
['__displayhook__', ... pozostałe nazwy zostały pominięte..., 'winver']
```

Wyniki przedstawione powyżej pochodzą z wersji 3.3 Pythona i celowo pominęliśmy w nich większość zwracanych nazw, ponieważ różnią się one nieco w zależności od wersji; aby uzyskać najbardziej aktualne wyniki, powinieneś samodzielnie uruchomić to polecenie we własnym systemie. W rzeczywistości w Pythonie istnieje obecnie 78 atrybutów, choć w zasadzie interesujących dla nas jest tylko 69, które nie mają wiodących podwójnych znaków podkreślenia (dwa podkreślenia zwykle oznaczają nazwy związane z interpreterem), lub 62, które nie mają wcale wiodących znaków podkreślenia (jeden znak podkreślenia zwykle oznacza

nieformalną nazwę prywatną) — swoją drogą jest to doskonała okazja do zastosowania listy składanej, którą omówiliśmy w poprzednim rozdziale:

```
>>> len(dir(sys))                                     # Liczba nazw w
module sys

78

>>> len([x for x in dir(sys) if not x.startswith('__')])    # Tylko nazwy bez
podwójnego podkreślenia

69

>>> len([x for x in dir(sys) if not x[0] == '_'])        # Tylko nazwy bez
podkreśleń

62
```

Aby dowiedzieć się, jakie atrybuty udostępniane są we wbudowanych typach obiektów, powinieneś wykonać funkcję `dir` na dowolnym literale (lub istniejącej instancji) pożądanego typu. Na przykład, aby zobaczyć atrybuty list i łańcuchów znaków, możemy przekazać do funkcji `dir` puste obiekty takich typów.

```
>>> dir([])

['__add__', '__class__', '__contains__', ...więcej..., 'append', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> dir('')

['__add__', '__class__', '__contains__', ...więcej..., 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Wynik wywołania funkcji `dir` dla dowolnego typu wbudowanego zawiera również listę atrybutów powiązanych z implementacją tego typu (z technicznego punktu widzenia, metod przeciążających operatory). Podobnie jak w modułach, wszystkie takie nazwy zaczynają się i kończą podwójnymi podkreśleniami, co pozwala na ich łatwe odróżnienie; w tym miejscu książki możesz je bezpiecznie zignorować (są one używane do programowania zorientowanego obiektowo). Na przykład istnieje 45 atrybutów list, ale tylko 11 z nich reprezentuje nazwane metody:

```
>>> len(dir([])), len([x for x in dir([]) if not x.startswith('__')])

(45, 11)

>>> len(dir('')), len([x for x in dir('') if not x.startswith('__')])

(76, 44)
```

W rzeczywistości, aby odfiltrować elementy o nazwach rozpoczynających się od podwójnego podkreślenia, które nie są nam potrzebne, utwórz taką samą listę składaną, ale wyświetl tylko atrybuty. Na przykład, aby wyświetlić listę atrybutów list i słowników w Pythonie 3.3, powinieneś wykonać następujące polecenia:

```
>>> [a for a in dir(list) if not a.startswith('__')]

['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']

>>> [a for a in dir(dict) if not a.startswith('__')]

['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
```

```
'setdefault', 'update', 'values']
```

Na pierwszy rzut oka może się wydawać, że to całkiem sporo kodu do wpisania jak na wyświetlenie listy atrybutów, ale na początku następnego rozdziału nauczyszmy się, jak opakować taki kod w funkcję wielokrotnego użytku, którą będziesz mógł zimportować, dzięki czemu nie będziesz musiał wpisywać jej ponownie:

```
>>> def dir1(x): return [a for a in dir(x) if not a.startswith('__')] #  
Zobacz część IV książki  
...  
>>> dir1(tuple)  
['count', 'index']
```

Warto zauważyć, że wbudowane atrybuty typu możemy również wyświetlić, przekazując do funkcji `dir` nazwę typu zamiast literala:

```
>>> dir(str) == dir('')                      # Ten sam wynik, niezależnie od tego,  
czy podamy nazwę, czy literał  
True  
>>> dir(list) == dir([])  
True
```

To rozwiązanie działa, ponieważ nazwy takie jak `str` i `list`, które kiedyś były funkcjami spełniającymi rolę konwerterów typów, obecnie są w Pythonie nazwami typów. Wywołanie jednej z nich powoduje wywołanie konstruktora, który generuje instancję określonego typu. Więcej informacji na temat konstruktorów oraz metod przeciążania operatorów zamieścimy podczas omawiania klas w szóstej części książki.

Wyniki działania funkcji `dir` stanowią jednak swego rodzaju zagadkę. Po jej uruchomieniu otrzymujemy co prawda listę nazw atrybutów, jednak nie mamy żadnych dodatkowych wskazówek na temat tego, co te nazwy oznaczają. Po takie dodatkowe informacje musimy zatem sięgnąć do kolejnego źródła dokumentacji.



Niektóre zintegrowane środowiska IDE do pracy w języku Python, w tym IDLE, posiadają wbudowane mechanizmy, które automatycznie wyświetlają listę atrybutów obiektów w jednym z okien, co może być postrzegane jako alternatywa dla polecenia `dir`. Na przykład środowisko IDLE wyświetla listę rozwijaną z atrybutami obiektu, gdy po nazwie obiektu wpiszesz kropkę i poczekasz przez moment lub naciśniesz klawisz `Tab`. Jest to jednak głównie mechanizm autouzupełniania, a nie źródło informacji. Więcej szczegółowych informacji na temat środowiska IDLE znajdziesz w rozdziale 3.

Notki dokumentacyjne — `_doc_`

Oprócz komentarzy ze znakami `#` Python obsługuje również dokumentację automatycznie dołączaną do obiektów i przechowuje ją do przeglądania w czasie działania programu. Z punktu widzenia składni takie komentarze są zakodowane jako ciągi znaków umieszczone na początku plików modułów czy definicji funkcji i klas, przed kodem wykonywalnym (można umieszczać przed nimi komentarze ze znakami `#`, wyłącznie z uniksowymi dyrektywami `shebang #!`). Python automatycznie wstawia zawartość takich łańcuchów znaków (znanych jako *notki dokumentacyjne* lub po prostu *notki docstrings*) do atrybutów `_doc_` odpowiednich obiektów.

Notki dokumentacyjne zdefiniowane przez użytkownika

Rozważmy na przykład poniższy plik, *docstrings.py*, w którym notki dokumentacyjne pojawiają się na początku pliku oraz na początku funkcji i klasy w niej się znajdującej. Użyłem tutaj komentarzy wielowierszowych w postaci bloku tekstu z potrójnymi cudzysłowami, jednak może je zastąpić dowolny inny rodzaj łańcuchów znaków: jednowierszowe komentarze ujęte w apostrofy lub cudzysłowy, takie jak użyte w definicji klasy, również są w porządku, ale nie pozwalają na tworzenie tekstu wielowierszowego. Nie omawialiśmy jeszcze instrukcji `def` i `class`, dlatego możesz w nich spokojnie zignorować wszystko z wyjątkiem łańcuchów znaków na początku definicji.

```
"""
Dokumentacja modułu
Tutaj znajduje się opis
"""

spam = 40

def square(x):
    """
    Dokumentacja funkcji square()
    Podnosi liczbę do kwadratu
    """

    return x **2                      # Kwadrat

class Employee:
    "dokumentacja klasy"
    pass

print(square(4))
print(square.__doc__)
```

Istotą tego protokołu dokumentacji jest to, żeby komentarze zostały zachowane do przejrzenia w atrybutach `__doc__` po zimportowaniu pliku. Aby zatem wyświetlić notki dokumentacyjne powiązane z modelem i jego obiektami, powinieneś zimportować plik i wyświetlić ich atrybuty `__doc__`, w których Python zapisał tekst.

```
>>> import docstrings
16
Dokumentacja funkcji square()
Podnosi liczbę do kwadratu
>>> print(docstrings.__doc__)
Dokumentacja modułu
Tutaj znajduje się opis
>>> print(docstrings.square.__doc__)

Dokumentacja funkcji square()
```

```
Podnosi liczbę do kwadratu
>>> print(docstrings.Employee.__doc__)
dokumentacja klasy
```

Warto zauważyc, że w przypadku chęci wyświetlenia notek dokumentacyjnych zazwyczaj będziemy chcieli użyć instrukcji `print`, w przeciwnym wypadku otrzymamy jeden łańcuch znaków z osadzonymi znakami nowych wierszy `\n`.

Notki dokumentacyjne można również dołączyć do *metod* klas (omówionych w szóstej części książki), jednak ponieważ są one po prostu instrukcjami `def` zagnieźdzonymi w instrukcjach `class`, nie są żadnym przypadkiem specjalnym. Aby pobrać notkę dokumentacyjną dla metody znajdującej się w klasie modułu, wystarczy rozszerzyć ścieżkę i przejść do metody przez klasę — `moduł.klasa.metoda.__doc__` (przykład notki dokumentacyjnej dla metody znajdziesz w rozdziale 29.).

Standardy i priorytety notek dokumentacyjnych

Jak wspominaliśmy wcześniej, powszechna praktyka zaleca dziś tworzenie komentarzy ze znakiem `#` tylko w dokumentacji pojedynczych wyrażeń, instrukcji lub małych grup poleceń. Notki dokumentacyjne są lepiej przystosowane do tworzenia dokumentacji na wyższym poziomie, obejmującej szerszą dokumentację funkcjonalną pliku, funkcji lub klasy, i dlatego stały się oczekiwana częścią programów napisanych w języku Python. To jednak tylko mechanizm, który nie zmienia w niczym faktu, że samą zawartość takich notek dokumentacyjnych musisz już napisać samodzielnie.

Choć niektóre firmy korzystające z języka Python i tworzące w nim oprogramowanie posiadają swoje własne, wewnętrzne zbiory wytycznych, to jednak nie istnieje żaden uniwersalny standard określający, co powinno się znaleźć w notkach dokumentacyjnych. Z biegiem czasu pojawiały się, co prawda, różne propozycje dotyczące języków znaczników i szablonów dla takich notek (na przykład HTML lub XML), jednak wszystko wskazuje na to, że w świecie Pythona się one nie przyjęły. Szczerze mówiąc, przekonanie programistów Pythona do tego, aby dokumentowali swój kod za pomocą ręcznie pisanej kodu HTML, nie nastąpi raczej w dającej się przewidzieć przyszłości... To już chyba by było dla nich zdecydowanie zbyt wiele, ale na szczęście nie dotyczy to samego dokumentowania kodu jako takiego.

Nie da się jednak ukryć, że tworzenie dokumentacji programu ma wśród wielu programistów zdecydowanie zbyt niski priorytet. Zazwyczaj, jeżeli w pobranym pliku źródłowym w ogóle znajdziesz jakiekolwiek komentarze, możesz się zaliczyć do szczęśliwców (a nawet coś bardziej, jeżeli komentarze znalezione w kodzie są precyzyjne i aktualne). Ja sam zachęcam jednak każdego do obfitego dokumentowania kodu — dokumentacja naprawdę jest bardzo istotną częścią dobrze napisanego programu. Jeżeli jednak nawet to robisz, główny problem polega na tym, że nie istnieje obecnie żaden standard dotyczący struktury notek dokumentacyjnych; jeżeli chcesz z nich korzystać, po prostu je tworzysz. Podobnie jak w przypadku pisania samego kodu, od Ciebie zależy to, jak będzie wyglądała dokumentacja i czy będzie aktualizowana, a najlepszym Twoim sprzymierzeńcem w tym zadaniu jest po prostu zdrowy rozsądek.

Wbudowane notki dokumentacyjne

Jak się okazuje, wbudowane moduły i obiekty Pythona wykorzystują podobne techniki do dołączania dokumentacji wykraczającej poza listę atrybutów zwracaną przez funkcję `dir`. Na przykład, aby wyświetlić opis wybranego, wbudowanego modułu, powinieneś go zimportować i wyświetlić jego atrybut `__doc__`.

```
>>> import sys
>>> print(sys.__doc__)
This module provides access to some objects used or maintained
```

by the interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

```
argv -- command line arguments; argv[0] is the script pathname if known  
path -- module search path; path[0] is the script directory, else ''  
modules -- dictionary of loaded modules  
...resztę tekstu pominieto...
```

Opisy funkcji, klas oraz metod wbudowanych modułów również dołączane są za pomocą atrybutów `__doc__`.

```
>>> print(sys.getrefcount.__doc__)  
getrefcount(object) -> integer  
  
Return the reference count of object. The count returned is generally  
one higher than you might expect, because it includes the (temporary)  
reference as an argument to getrefcount().
```

W notkach dokumentacyjnych możesz również znaleźć dodatkowe informacje na temat wbudowanych funkcji.

```
>>> print(int.__doc__)  
int(x[, base]) -> integer  
  
Convert a string or number to an integer, if possible. A floating  
point argument will be truncated towards zero (this does not include a  
...resztę tekstu pominieto...  
  
>>> print(map.__doc__)  
map(func, *iterables) --> map object  
  
Make an iterator that computes the function using arguments from  
each of the iterables. Stops when the shortest iterable is exhausted.
```

Przeglądając notki dokumentacyjne, możemy znaleźć wiele informacji na temat wbudowanych narzędzi. Nie musimy tego jednak robić — funkcja `help`, do której zaraz przejdziemy, robi to za nas automatycznie.

PyDoc — funkcja `help`

Notki dokumentacyjne okazały się na tyle przydatne, że w Pythonie dostępne jest teraz narzędzie ułatwiające ich wyświetlanie. Standardowe narzędzie `PyDoc` to program w Pythonie, potrafiący łączyć ze sobą notki dokumentacyjne i powiązane z nimi informacje strukturalne oraz formatować je w ładnie zaaranżowane raporty różnego typu. Dodatkowe narzędzia służące do ekstrakcji i formatowania notek dokumentacyjnych dostępne są na licencji open source (w tym narzędzia obsługujące tekst ustrukturyzowany — więcej na ten temat można znaleźć w internecie), natomiast moduł `PyDoc` jest częścią biblioteki standardowej Pythona.

Istnieje wiele sposobów uruchamiania pakietu `PyDoc`, włącznie z użyciem opcji wywołania z poziomu wiersza poleceń, które pozwalają zapisać wynikową dokumentację do późniejszego przejrzenia (ich opis znajdziesz zarówno w dokumentacji biblioteki Pythona, jak i w dalszej

części tego rozdziału). Prawdopodobnie najbardziej znane interfejsy pakietu PyDoc to wbudowana funkcja `help` oraz interfejsy graficzny i webowy wykorzystujący strony HTML. Funkcję `help` omówiliśmy krótko w rozdziale 4.; wykorzystuje ona moduł PyDoc w celu wygenerowania prostego raportu tekstowego dla dowolnego obiektu będącego argumentem jej wywołania. W tym trybie tekst pomocy wygląda jak strona podręcznika `man` w systemach uniksopodobnych i w praktyce wyświetlanie kolejnych stron pomocy działa tak samo — naciśnij spację, aby przejść do następnej strony, klawisz *Enter*, aby przejść do następnego wiersza, a klawisz *Q*, aby zakończyć przeglądanie i powrócić do konsoli Pythona:

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount in module sys:
getrefcount(...)

    getrefcount(object) -> integer

    Return the reference count of object. The count returned is generally
    one higher than you might expect, because it includes the (temporary)
    reference as an argument to getrefcount().
```

Warto zauważyć, że aby wywołać funkcję `help`, wcale nie trzeba importować modułu `sys`. Aby jednak w ten sposób uzyskać pomoc dotyczącą samego modułu `sys`, trzeba go zaimportować, ponieważ funkcja `help` oczekuje przekazania referencji do obiektu. W wersjach 3.3 i 2.7 Pythona możesz wyświetlić pomoc dla modułu, który nie został zaimportowany, umieszczając jego nazwę w apostrofach — na przykład `help('re')`, `help('email.message')` — ale działanie tego i innych trybów może się różnić w zależności od wersji Pythona.

W przypadku większych obiektów, takich jak moduły i klasy, ekran pomocy jest podzielony na wiele sekcji, których preambuły pokazano tutaj. Aby zobaczyć pełny raport, powinieneś uruchomić to polecenie w sesji interaktywnej (w przykładzie korzystam z niego w wersji 3.3):

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

MODULE REFERENCE
    http://docs.python.org/3.3/library/sys
    ...pozostała część została pominięta...

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.
    ...pozostała część została pominięta...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None
    ...pozostała część została pominięta...
```

```
DATA
    __stderr__ = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='cp4...
    __stdin__ = <_io.TextIOWrapper name='<stdin>' mode='r' encoding='cp437...
    __stdout__ = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='cp4...
...pozostała część została pominięta...

FILE
(built-in)
```

Część informacji z tego raportu to zawartość notek dokumentacyjnych, a część (na przykład wzorce wywoływania funkcji) to informacje strukturalne zbierane przez PyDoc automatycznie przy okazji inspekcji zawartości poszczególnych obiektów, o ile są one dostępne.

Funkcję `help` można również wywołać na wbudowanych funkcjach, metodach oraz typach. Sposób użycia może się nieco różnić w zależności od wersji Pythona, ale aby otrzymać pomoc dotyczącą *typu wbudowanego*, powinieneś użyć albo nazwy typu (na przykład `dict` dla słownika, `str` dla łańcucha znaków, `list` dla listy), albo obiektu danego typu (na przykład `{}`, `' '`, `[]`), albo metody danego obiektu bądź nazwy typu (na przykład `str.join`, `'s'.join`)^[1]. W wyniku działania otrzymamy długi tekst opisujący wszystkie metody dostępne dla określonego typu.

```
>>> help(dict)
Help on class dict in module builtins:
class dict(object)
| dict() -> new empty dictionary.
| dict(mapping) -> new dictionary initialized from a mapping object's
...pozostała część została pominięta...

>>> help(str.replace)
Help on method_descriptor:
replace(...)
    S.replace (old, new[, count]) -> str
    Return a copy of S with all occurrences of substring
    ...pozostała część została pominięta...

>>> help(''.replace)
...rezultat podobny do poprzedniego polecenia...

>>> help(ord)
Help on built-in function ord in module builtins:
ord(...)
    ord(c) -> integer
    Return the integer ordinal of a one-character string.
```

Funkcja `help` na naszych modułach działa tak samo dobrze jak na tych wbudowanych. Poniżej widać raport dotyczący utworzonego wcześniej pliku `docstrings.py`. Ponownie część tekstu to

notki dokumentacyjne, a część to informacje pobierane automatycznie dzięki inspekcji struktury obiektu.

```
>>> import docstrings
>>> help(docstrings.square)
Help on function square in module docstrings:
square(x)

    Dokumentacja funkcji square()
        Podnosi liczbę do kwadratu

>>> help(docstrings.Employee)
Help on class Employee in module docstrings:
class Employee(builtins.object)
| dokumentacja klasy
|
| Data descriptors defined here:
|   ...resztę tekstu pominięto...
>>> help(docstrings)
Help on module docstrings:

NAME
    docstrings

DESCRIPTION
    Dokumentacja modułu
    Tutaj znajduje się opis

CLASSES
    builtins.object
        Employee
            class Employee(builtins.object)
                | dokumentacja klasy
                |
                | Data descriptors defined here:
                |   ...resztę tekstu pominięto...

FUNCTIONS
    square(x)

        Dokumentacja funkcji square()
            Podnosi liczbę do kwadratu

DATA
```

```
spam = 40
FILE
c:\code\docstrings.py
```

PyDoc – raporty HTML

Wyświetlanie w formie tekstowej wyników działania funkcji `help` jest w wielu przypadkach w zupełności wystarczające, zwłaszcza podczas pracy w sesji interaktywnej. Jednak użytkownikom, którzy przyzwyczaili się do bogatszych form prezentacji, takie rozwiązanie może wydawać się nieco prymitywne. W tej sekcji omówimy mechanizmy pakietu PyDoc oparte na języku HTML, które pozwalają na renderowanie dokumentacji modułu w postaci gotowej do przeglądania w przeglądarce internetowej. W wersji 3.3 Pythona zmienił się sposób uruchamiania tej funkcji:

- *Przed wersją 3.3* Python był dostarczany z prostym klientem graficznym GUI do przesyłania żądań wyszukiwania. Klient uruchamiał przeglądarkę internetową, aby wyświetlić dokumentację utworzoną przez automatycznie uruchamiany serwer lokalny.
- *W wersji 3.3* poprzedni klient GUI został zastąpiony schematem interfejsu dla wszystkich przeglądarek, który łączy zarówno wyszukiwanie, jak i wyświetlanie na stronie internetowej komunikującej się z automatycznie uruchamianym serwerem lokalnym.
- *Python w wersji 3.2* znajduje się gdzieś na granicy między tymi dwoma rozwiązaniami i obsługuje zarówno oryginalny schemat klienta GUI, jak i nowszy tryb dla wszystkich przeglądarek obowiązujący od wersji 3.3.

Ponieważ odbiorcami tej książki są użytkownicy zarówno najnowszych i najlepszych, jak i starszych, ale nadal dobrych wersji Pythona, omówimy tutaj oba schematy. Pamiętaj jednak, że różnice między tymi schematami dotyczą tylko najwyższego poziomu ich interfejsów użytkownika. Wyświetlane przez nie dokumenty są prawie identyczne i w obu systemach PyDoc może być również używany do generowania zarówno tekstu w konsoli, jak i plików HTML do późniejszego przeglądania w dowolny sposób.

Python 3.2 i nowsze wersje: tryb PyDoc dla wszystkich przeglądarek

Począwszy od wersji 3.3 Pythona, oryginalny tryb klienta GUI PyDoc, obecny w wersjach 2.x i wcześniejszych 3.x, nie jest już dostępny. Ten tryb jest obecny w Pythonie 3.2 ze skrótem `Module Docs` w menu `Start` systemu Windows 7 i wcześniejszych oraz z poziomu wiersza poleceń konsoli za pomocą polecenia `pydoc -g`. W wersji 3.2 ten tryb GUI został ogłoszony jako przestarzały, chociaż działa zupełnie dobrze i bez żadnych problemów na mojej maszynie testowej.

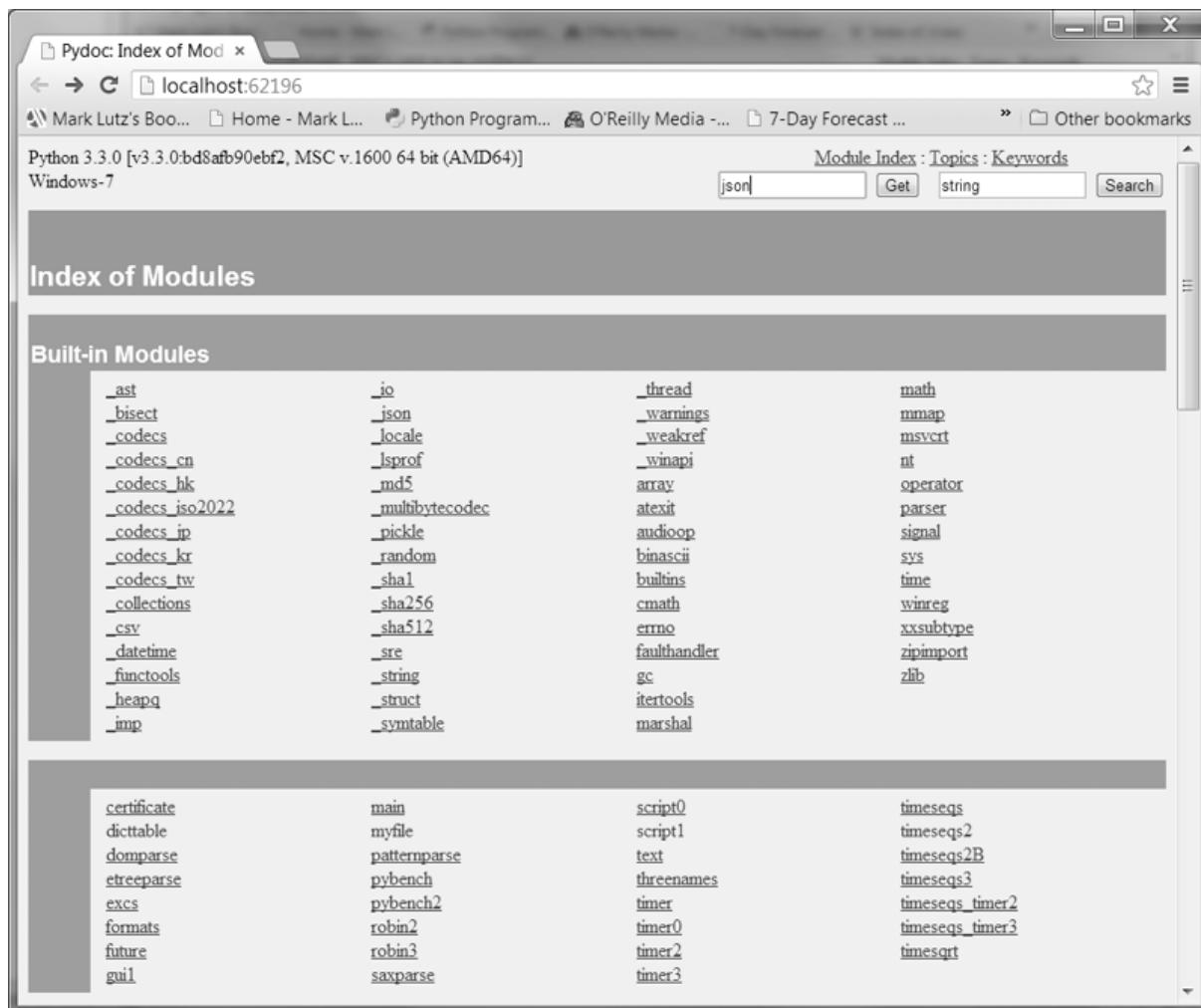
W wersji 3.3 tryb ten został jednak całkowicie usunięty i zastąpiony poleceniem `pydoc -b` uruchamianym z poziomu wiersza poleceń konsoli, które zamiast klienta GUI uruchamia zarówno lokalnie działający serwer dokumentacji, jak i przeglądarkę internetową działającą jako klient do wyszukiwania, a także wyświetlania stron pomocy. Przeglądarka jest początkowo otwierana na stronie indeksu modułów, który ma nieco rozszerzoną funkcjonalność. Istnieją również dodatkowe sposoby korzystania z PyDoc (np. zapisywanie strony HTML w pliku do późniejszego przeglądania, jak opisano wcześniej), więc jest to stosunkowo niewielka zmiana funkcjonalności.

Aby uruchomić nowszy tryb PyDoc przeznaczony tylko dla przeglądarki sieciowej w Pythonie 3.2 i nowszych, wystarczy wydać z poziomu wiersza poleceń jedno z polecień przedstawionych poniżej; wszystkie z nich używają opcji `-m` dla wygody, aby zlokalizować plik modułu PyDoc w ścieżce wyszukiwania modułów. Pierwszy przykład zakłada, że Python znajduje się w ścieżce systemowej; drugi wykorzystuje nowy program uruchamiający Pythona 3.3 (dla systemu

Windows); a trzeci podaje pełną ścieżkę do Pythona, jeżeli pozostałe dwa schematy nie działają. Więcej informacji na temat opcji `-m` znajdziesz w załączniku A, a w dodatku B omówiono program uruchamiający Pythona dla systemu Windows.

```
c:\code> python -m pydoc -b
Server ready at http://localhost:62135/
Server commands: [b]rowser, [q]uit
server> q
Server stopped
c:\code> py -3 -m pydoc -b
Server ready at http://localhost:62144/
Server commands: [b]rowser, [q]uit
server> q
Server stopped
c:\code> C:\python33\python -m pydoc -b
Server ready at http://localhost:62153/
Server commands: [b]rowser, [q]uit
server> q
Server stopped
```

Niezależnie jednak od użytego polecenia efektem będzie uruchomienie pakietu PyDoc jako lokalnie działającego serwera WWW na dedykowanym (ale domyślnie nieużywanym) porcie i wyświetlenie przeglądarki internetowej, która będzie działać jako *klient*, wyświetlając stronę z linkami do dokumentacji wszystkich modułów dostępnych do zainportowania w ścieżce wyszukiwania modułów (z katalogiem, z którego został uruchomiony PyDoc, włącznie). Strona główna interfejsu pakietu PyDoc została pokazana na rysunku 15.1.



Rysunek 15.1. Strona główna interfejsu pakietu PyDoc dla wszystkich przeglądarek w Pythonie 3.2 i nowszych. Od Pythona 3.3 zastępuje ona klienta GUI dostępnego we wcześniejszych wersjach

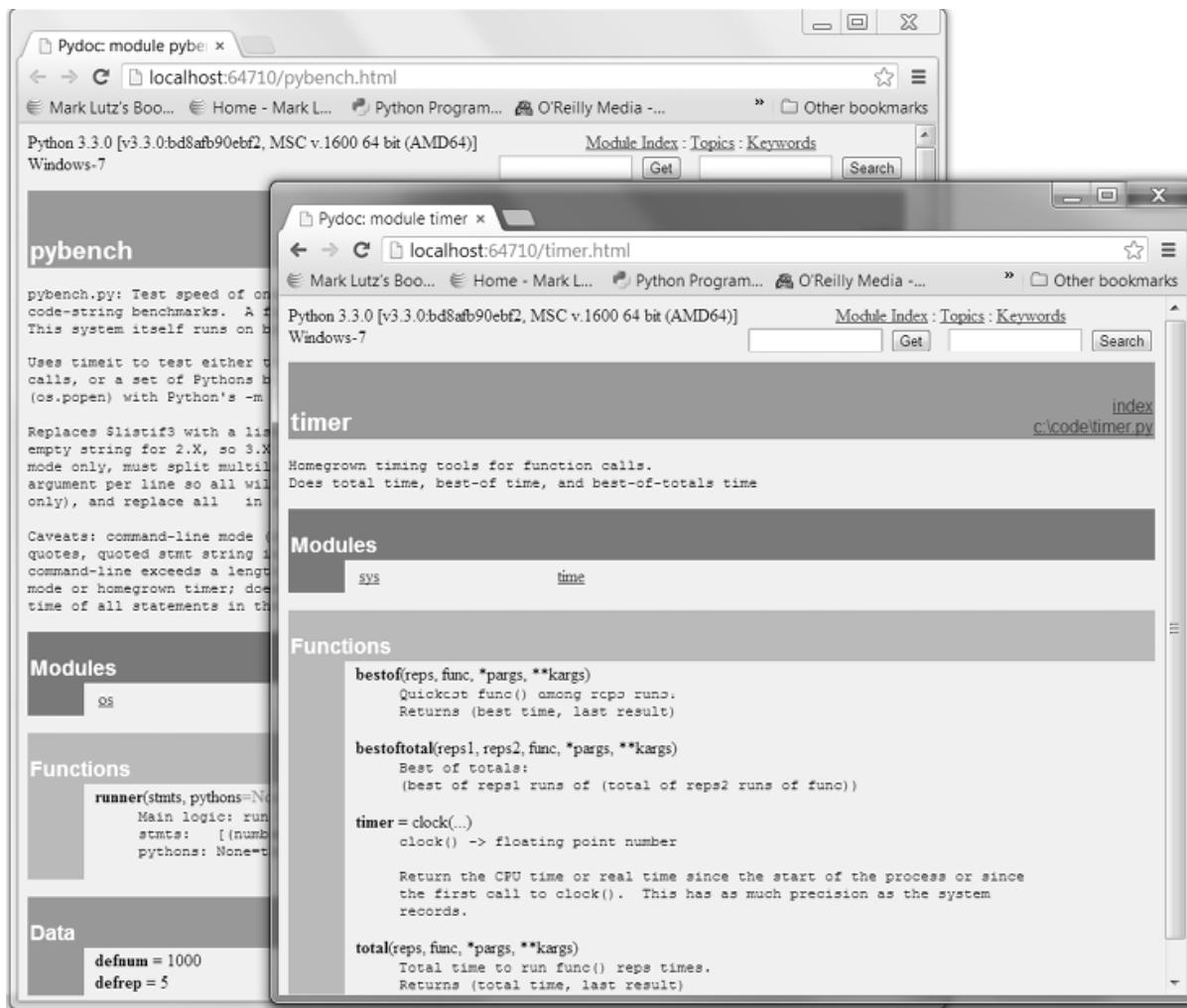
Oprócz indeksu modułów strona internetowa PyDoc zawiera również pola wyszukiwania, gdzie możesz wpisać nazwę żądanego modułu (pole i przycisk *Get*) i wyszukać powiązane wpisy (pole i przycisk *Search*), które odpowiadają opcjom wyszukiwania w kliencie GUI poprzedniego interfejsu. Możesz również kliknąć łącza wyświetlane na tej stronie, aby przejść do indeksu modułów (łącze *Module Index*; wyświetla stronę główną), tematów (łącze *Topics*; wyświetla listę ogólnych zagadnień języka Python) i słów kluczowych (łącze *Keywords*; wyświetla przegląd instrukcji i niektórych wyrażeń).

Zwróć uwagę, że strona indeksu przedstawiona na rysunku 15.1 zawiera odniesienia zarówno do *modułów*, jak i *skryptów* znajdujących się w bieżącym katalogu, z którego PyDoc została uruchomiona. Pakiet PyDoc jest głównie przeznaczony do dokumentowania modułów, które można importować, ale czasem można go także wykorzystać do pokazania dokumentacji skryptów. Aby można było wyświetlić dokumentację, wybrany plik musi zostać zimportowany, a jak wspominaliśmy już wcześniej, importowanie uruchamia kod zawarty w pliku, jednak moduły zwykle zawierają po prostu definicje narzędzi, więc zazwyczaj nie ma to większego znaczenia.

Gdy zechcesz wyświetlić dokumentację dla pliku skryptu najwyższego poziomu, okno powłoki, z którego uruchomiłeś PyDoc, będzie służyło jako standardowe wejście i wyjście skryptu dla

każdej interakcji użytkownika. Efektem netto jest to, że strona dokumentacji skryptu pojawi się po jego uruchomieniu, a po wyświetleniu pojawi się w oknie powłoki. W przypadku niektórych skryptów może to jednak działać lepiej niż dla innych; na przykład interaktywne wprowadzanie danych może dziwnie przeplatać się z informacjami pochodząymi z samego serwera PyDoc.

Po przejściu ze strony głównej, przedstawionej na rysunku 15.1, strony dokumentacji dla poszczególnych modułów są zasadniczo takie same zarówno w nowszym trybie przeglądarki, jak i we wcześniejszym schemacie klienta GUI, oprócz dodatkowych pól wyszukiwania widocznych w nowej wersji na górze strony. Na rysunku 15.2 pokazano przykładowe strony dokumentacji, otwarte na dwóch zdefiniowanych przez użytkownika modułach, które napiszemy w następnej części tej książki podczas omawiania zagadnień związanych z pomiarami czasu i wydajności w rozdziale 21. W obu schematach strony dokumentacji zawierają automatycznie tworzone hiperłącza, które pozwalają na przeglądanie dokumentacji powiązanych komponentów w aplikacji, na przykład łączy do stron importowanych modułów.



Rysunek 15.2. Przykładowe strony dokumentacji, wyświetcone w module PyDoc Pythona 3.2, otwarte na dwóch zdefiniowanych przez użytkownika modułach, które napiszemy w następnej części tej książki (w rozdziale 21.)

Ze względu na podobieństwo wyświetlanych stron informacje przedstawione w kolejnej sekcji, dotyczącej pakietu PyDoc w wersjach wcześniejszych niż 3.2, oraz zrzuty ekranu w dużej

mierze mają zastosowanie również do wersji nowszych niż 3.2, zatem powinieneś się z nimi zapoznać, nawet jeśli używasz nowszego Pythona. Pakiet PyDoc w wersji 3.3 po prostu odcina starszego, pośredniego klienta GUI w wersji wcześniejszej niż 3.2, zachowując jednocześnie przeglądarkę i serwer.

Pakiet PyDoc w Pythonie 3.3 nadal obsługuje jednak poprzednie, konsolowe tryby użytkowania. Na przykład polecenia `pydoc -p numerportu` można użyć do ustawienia portu serwera PyDoc, a polecenie `pydoc -w nazwamodułu` nadal zapisuje dokumentację modułu w pliku HTML o nazwie `nazwamodułu.html` do późniejszego przeglądania. Tylko tryb klienta GUI `pydoc -g` został usunięty i zastąpiony przez `pydoc -b`. Możesz również uruchomić pakiet PyDoc, aby wygenerować dokumentację w postaci zwykłego tekstu (jego wygląd, przypominający wygląd stron podręcznika `man`, pokazaliśmy już wcześniej w tym rozdziale) — polecenie przedstawione poniżej odpowiada wywołaniu polecenia `help` w interaktywnej sesji Pythona:

```
c:\code> py -3 -m pydoc timeit          # Wyświetlenie pomocy z poziomu
wiersza poleceń

c:\code> py -3
>>> help("timeit")                   # Wyświetlenie pomocy w sesji
interaktywnej Pythona
```

Ze względu na jego interaktywność najlepszym rozwiązaniem jest samodzielne wypróbowanie webowego interfejsu pakietu PyDoc, dlatego nie będziemy tutaj zbytnio się rozwodzić nad szczegółami jego użytkowania; dodatkowe informacje i opisy opcji wiersza poleceń znajdziesz w dokumentacji Pythona. Warto również zwrócić uwagę, że zarówno serwer PyDoc, jak i niemal cała funkcjonalność związana ze stronami dla przeglądarki zostały utworzone za pomocą modułów standardowej biblioteki Pythona (na przykład `webbrowser`, `http.server`). Aby dowiedzieć się, jak to zostało zrobione, i poszukać inspiracji do własnych rozwiązań, powinieneś zjrzeć do kodu źródłowego pakietu PyDoc, który znajdziesz w pliku `pydoc.py`.

Zmiana kolorów PyDoc

W papierowej wersji tej książki nie będziesz w stanie tego sprawdzić, ale jeśli masz e-booka lub uruchomisz pakiet PyDoc na żywo, zauważysz, że interfejs jest wyświetlany w kolorach, które mogą Ci się podobać lub nie. Niestety, obecnie nie ma łatwego sposobu na dostosowanie kolorów webowego interfejsu PyDoc. Są one zapisane głęboko w kodzie źródłowym i nie mogą być przekazywane jako argumenty do funkcji lub wiersza poleceń ani zmieniane w plikach konfiguracyjnych lub zmiennych globalnych w samym module PyDoc.

Nie zmienia to jednak w niczym faktu, że w programach open source zawsze możesz zmienić kod źródłowy — pakiet PyDoc zapisany jest w pliku `pydoc.py` w standardowej bibliotece Pythona, czyli w katalogu `C:\Python3x\Lib` w systemie Windows. Kolory interfejsu są zakodowanymi na stałe ciągami szesnastkowych wartości RGB osadzonych w różnych miejscach kodu. Na przykład ciąg znaków '`#eeaa77`' definiuje 3-bajtową (24-bitową) wartość, składającą się z trzech liczb szesnastkowych, reprezentujących jednobajtowe (8-bitowe) wartości poziomu dla każdego z trzech kolorów składowych RGB: czerwonego, zielonego i niebieskiego (dziesięćte 238, 170 i 119), co po złożeniu daje odcień koloru pomarańczowego, używanego na banerach z nazwami funkcji. Ciąg znaków '`#ee77aa`' w podobny sposób definiuje ciemnoróżowy kolor używany w dziewięciu innych miejscach, w tym na banerach na stronach indeksów i klas.

Aby dopasować kolory do własnych upodobań, wyszukaj ciągi znaków reprezentujące wartości kolorów i zamień je na swoje preferencje. W środowisku IDLE definicje kolorów możesz znaleźć po wybraniu z menu polecenia `Edit/Find` (edytuj/znajdź) i wpisaniu wyrażenia regularnego `#\w{6}`, które zgodnie z regułami składni modułu `re` Pythona dopasowuje ciągi składające się z sześciu znaków alfanumerycznych występujących po znaku `#`; więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji biblioteki.

Aby wybrać nowe kolory, możesz posłużyć się jednym z wielu programów pozwalających na wskazanie koloru i odczytanie jego wartości w standardzie RGB; przykłady z książki obejmują również skrypt *setcolor.py*, który robi to samo. W mojej wersji pakietu PyDoc zamieniłem wszystkie #ee77aa na #008080 (turkusowy), aby pozbyć się tego irytującego, ciemnego różu. Zamiana koloru #ffc8d8 na #c0c0c0 (szary) działa podobnie dla jasnoróżowego tła dokumentów klasy.

Taka operacja nie jest jednak przeznaczona dla osób niecierpliwych — plik PyDoc ma obecnie 2600 wierszy, ale za to jego modyfikowanie to naprawdę porządne ćwiczenie w utrzymywaniu kodu. Zachowaj ostrożność przy zamianie kolorów takich jak #ffffff i #000000 (biały i czarny), i najpierw wykonaj kopię zapasową pliku *pydoc.py*, aby mieć do czego wrócić w razie awarii. Ten plik korzysta z narzędzi, których jeszcze nie poznaliśmy, ale możesz bezpiecznie zignorować resztę kodu podczas wprowadzania zmian w kolorach.

Koniecznie obserwuj zmiany w konfiguracji pakietu PyDoc w kolejnych wersjach; wydaje się on być jednym z głównych kandydatów do modyfikacji. W rzeczywistości już trwają nad nim prace: zgłoszenie 10716 na liście poprawek Pythona ma na celu zwiększenie możliwości dostosowania PyDoc do potrzeb użytkownika poprzez zaimplementowanie obsługi arkuszy stylów CSS. Jeżeli się to powiedzie, pozwoli użytkownikom na wybór kolorów i innych opcji wyświetlania w zewnętrznych plikach CSS zamiast modyfikowania kodu źródłowego pakietu PyDoc.

Z drugiej strony będzie to wymagało od użytkowników pakietu PyDoc biegłości w posługiwaniu się kodem CSS, niestety posiadającym własną, nietrywialną strukturę, której wielu użytkowników Pythona może nie rozumieć w stopniu wystarczającym do wprowadzania modyfikacji. Gdy pisałem te słowa, proponowany plik PyDoc CSS zawierał już ponad 230 wierszy kodu mocno niezrozumiałego dla użytkowników, którzy nie mają doświadczenia w programowaniu stron internetowych (i nie wydaje się rozsądne, aby mieli się tego uczyć tylko po to, aby mieć możliwość dostosowania pakietu PyDoc do własnych potrzeb!).

Dzisiejszy PyDoc obsługuje już arkusze stylów CSS, które oferują pewne opcje dostosowywania, ale tylko w pewnym zakresie. Dopóki to nie zostanie rozwiązane, bezpośrednie zmiany w kodzie wydają się najlepszą opcją dostosowania. W każdym razie zagadnienia związane z arkuszami stylów CSS znacznie wykraczają poza zakres tej książki o języku Python — na szczęście w internecie dostępnych jest wiele materiałów na ten temat; znajdziesz tam również informacje o przyszłych wersjach języka Python i planowanym rozwoju pakietu PyDoc.

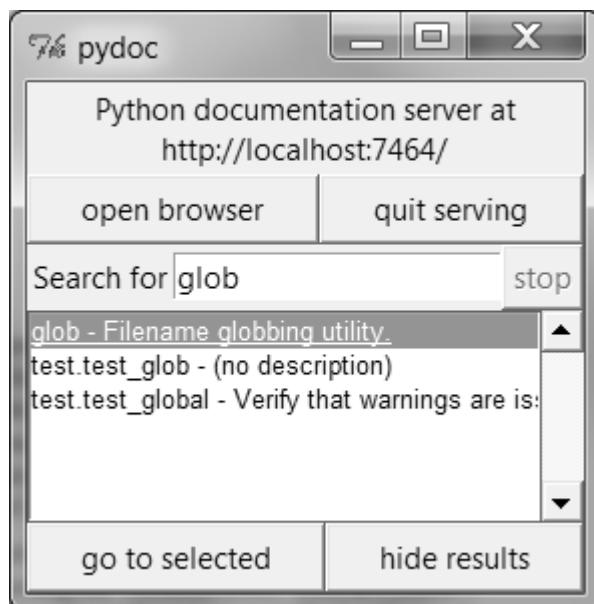
Python 3.2 i wersje wcześniejsze: klient GUI

Dla użytkowników korzystających z wersji 3.2 i wcześniejszych w tej sekcji omówimy oryginalnego klienta PyDoc GUI. Przedstawione zagadnienia opierają się na podstawach omówionych w poprzednich podrozdziałach, których nie będziemy tutaj powtarzać, zatem jeżeli coś będzie dla Ciebie niejasne, to pamiętaj, aby zajrzeć do poprzedniej sekcji (zwłaszcza gdy używasz starszego Pythona).

Jak wspomniano, w Pythonie 3.2 pakiet PyDoc daje użytkownikowi do dyspozycji interfejs graficzny (prosty, lecz jednocześnie przenośny skrypt oparty na Pythonie i module *tkinter*), a także serwer dokumentacji. Żądania z klienta są kierowane do serwera, który generuje raporty wyświetlane w przeglądarce internetowej. Cały proces jest w dużej mierze automatyczny, a rola użytkownika sprowadza się do przesyłania zapytań.

Aby uruchomić PyDoc w tym trybie, zwykle najpierw musisz uruchomić graficzny interfejs użytkownika silnika wyszukiwarki, pokazany na rysunku 15.3. Możesz to zrobić albo wybierając opcję *Module Docs* z menu *Python* dostępnego pod przyciskiem *Start* w systemie Windows, albo uruchamiając skrypt *pydoc.py* znajdujący się w katalogu biblioteki standardowej Pythona *Lib* w systemie Windows (powinieneś uruchomić go z opcją -g; możesz również użyć opcji -m, aby uniknąć konieczności wpisywania ścieżki do skryptu).

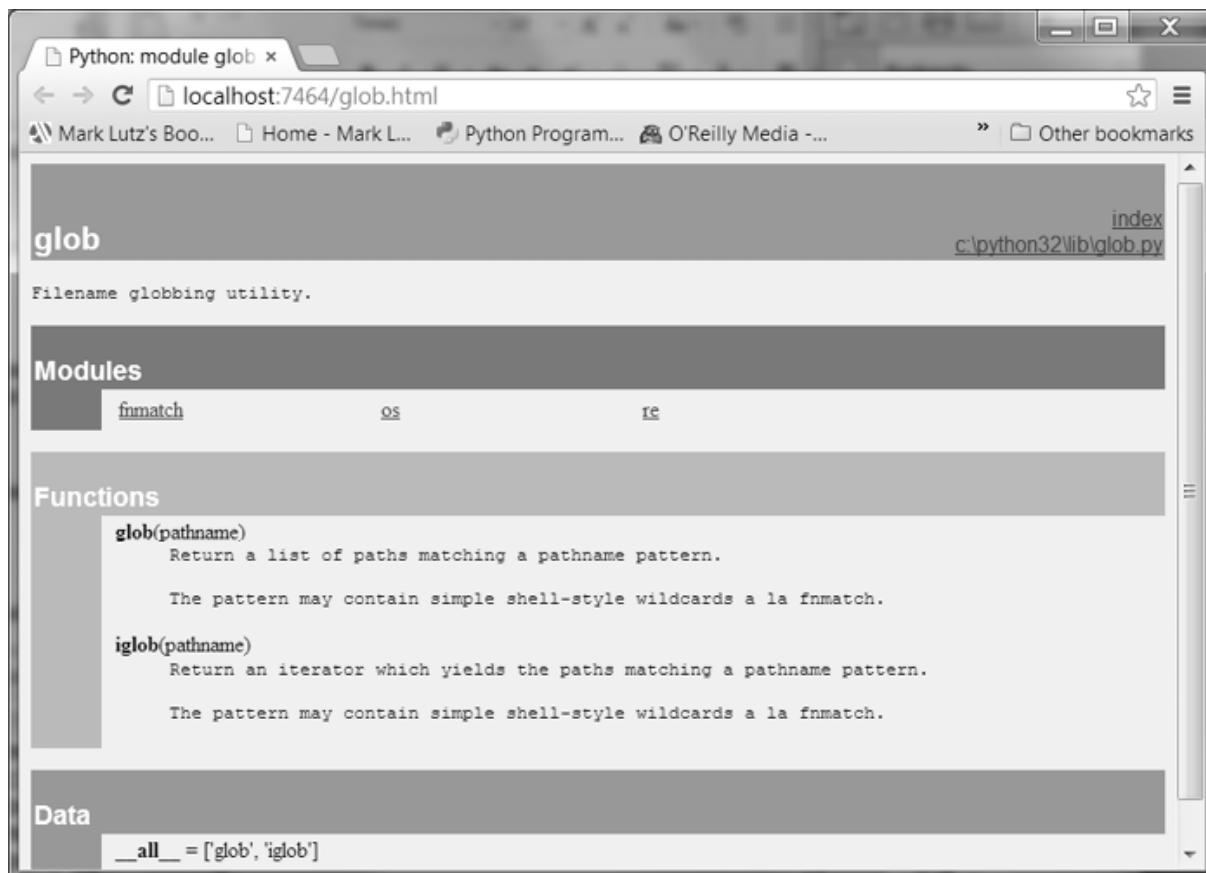
```
c:\code> c:\python32\python -m pydoc -g          # Wywołanie z pełną ścieżką  
do Pythona  
c:\code> py -3.2 -m pydoc -g                      # Wywołanie z użyciem  
launchera (dla wersji 3.3+)
```



Rysunek 15.3. Graficzny interfejs użytkownika silnika wyszukiwarki w Pythonie 3.2 i wersjach wcześniejszych; wpisz nazwę szukanego modułu, naciśnij Enter, wybierz moduł, a następnie naciśnij przycisk „go to selected” (lub pomiń nazwę modułu i kliknij „open browser”, aby wyświetlić listę wszystkich dostępnych modułów)

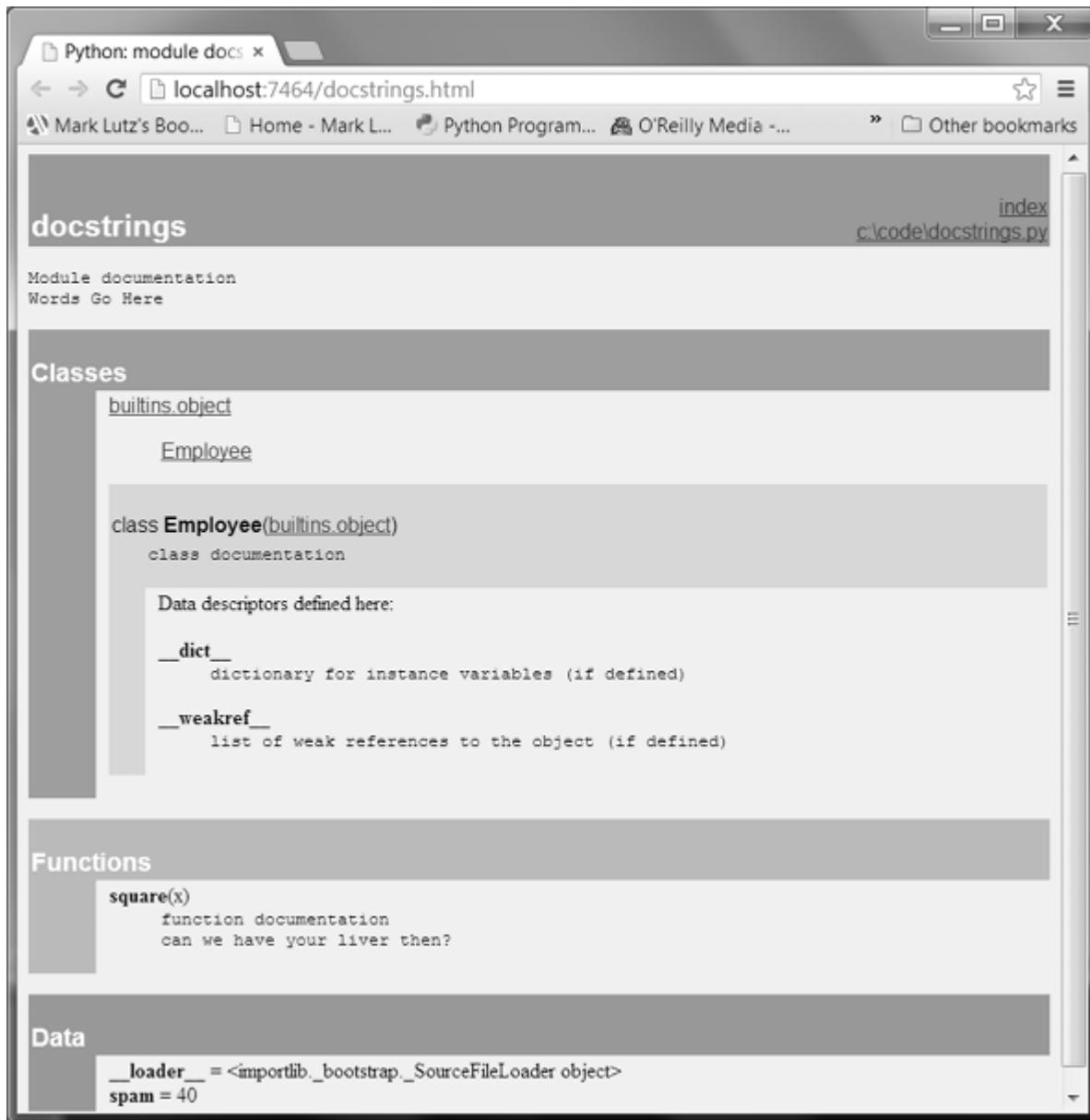
Wpisz nazwę modułu, który Cię interesuje, i naciśnij klawisz *Enter*. PyDoc przejdzie ścieżkę wyszukiwania modułów (`sys.path`), szukając żądanego modułu i odniesień do niego.

Po znalezieniu poszukiwanego modułu zaznacz go i naciśnij przycisk *go to selected* (przejdz do zaznaczonego modułu). PyDoc uruchomi przeglądarkę internetową, w której wyświetli raport wygenerowany w formacie HTML. Na rysunku 15.4 pokazano informacje wyświetlane przez PyDoc dla wbudowanego modułu `glob`. Zwróć uwagę na odnośniki (łącza) w części *Modules* takiej strony. Możesz je kliknąć, aby przejść na strony PyDoc poświęconej powiązanym, zimportowanym modułom. W przypadku większych stron PyDoc generuje również odnośniki do poszczególnych sekcji dokumentu.



Rysunek 15.4. Po znalezieniu modułu w graficznym interfejsie użytkownika z rysunku 15.3 i naciśnięciu przycisku „go to selected” dokumentacja modułu jest generowana w formacie HTML i wyświetlana w oknie przeglądarki internetowej

Podobnie jak w przypadku funkcji `help`, interfejs graficzny klienta PyDoc pozwala na wyszukiwanie dokumentacji zarówno modułów wbudowanych, jak i modułów zdefiniowanych przez użytkownika. Na rysunku 15.5 przedstawiono stronę wygenerowaną dla naszego modułu `docstrings.py`, który utworzyliśmy wcześniej.



Rysunek 15.5. PyDoc tworzy strony dokumentacji dla modułów wbudowanych oraz tworzonych przez użytkownika, znajdujących się w ścieżce wyszukiwania. Na rysunku pokazano stronę modułu zdefiniowanego przez użytkownika, na której znajdują się wszystkie notki dokumentacyjne pobrane z pliku źródłowego

Upewnij się, że katalog zawierający żądany moduł znajduje się na ścieżce wyszukiwania importowanych modułów. Jak wspominaliśmy, PyDoc musi mieć możliwość zimportowania pliku, aby wyświetlić jego dokumentację. Obejmuje to bieżący katalog roboczy — PyDoc może nie sprawdzić katalogu, z którego został uruchomiony (co prawdopodobnie nie ma żadnego znaczenia, gdy zostanie uruchomiony z poziomu menu *Start* systemu Windows), więc do poprawnego działania może okazać się konieczna modyfikacja ustawień zmiennej **PYTHONPATH**. W wersjach 3.2 i 2.7 musiało dodać „.” do zmiennej **PYTHONPATH**, aby graficzny klient pakietu PyDoc mógł przeglądać zawartość katalogu, z którego został uruchomiony z wiersza poleceń:

```
c:\code> set PYTHONPATH=.;%PYTHONPATH%
```

```
c:\code> py -3.2 -m pydoc -g
```

Taka modyfikacja była również wymagana, aby w wersji 3.2 wyświetlić zawartość bieżącego katalogu w nowym trybie `pydoc -b` (dla wszystkich przeglądarek). Warto jednak zauważać, że Python 3.3 automatycznie dodaje katalog „.” na swojej liście wyszukiwania, więc nie musimy już ręcznie modyfikować ustawień ścieżek, aby przeglądać pliki w katalogu, z którego PyDoc został uruchomiony — jest to co prawda niewielka, ale z pewnością godna uwagi poprawka.

PyDoc można dostosować do własnych wymagań i uruchamiać na różne sposoby, których nie będziemy tutaj omawiać. Więcej informacji na ten temat możesz znaleźć w dokumentacji Pythona. Najważniejszą kwestią, jaką powinieneś zapamiętać, jest to, że PyDoc tworzy dla Ciebie dokumentację programów całkowicie „gratis” — jeżeli zdecydujesz się w swoich programach używać notek dokumentacyjnych, PyDoc wykona za Ciebie całą pracę związaną z ich zbieraniem i formatowaniem w sposób odpowiedni do wyświetlania. PyDoc udostępnia dokumentację dla obiektów takich, jak funkcje i moduły oraz ułatwia dostęp do dokumentacji średniego poziomu dla takich narzędzi. Raporty z PyDoc są bardziej użyteczne od surowych list atrybutów i mniej wyczerpujące od standardowych podręczników czy dokumentacji.

PyDoc można również uruchomić, aby zapisać dokumentację HTML modułu w pliku do późniejszego przeglądania lub drukowania; wskazówki, jak to zrobić, znajdziesz w poprzedniej sekcji. Powinieneś również pamiętać, że PyDoc może nie działać dobrze, jeżeli zostanie uruchomiony na skryptach odczytujących dane ze standardowego wejścia — PyDoc importuje moduł docelowy w celu sprawdzenia jego zawartości i może nie mieć połączenia ze standardowym strumieniem wejścia, gdy zostanie uruchomiony w trybie GUI, szczególnie jeżeli zrobisz to za pomocą menu *Start* systemu Windows. Moduły, które można importować bez konieczności natychmiastowego wprowadzania danych wejściowych, zawsze będą poprawnie działać z PyDoc. Zobacz także uwagi z poprzedniej sekcji dotyczące skryptów w trybie `-b` PyDoc w wersji 3.2 i nowszych; uruchamianie trybu GUI PyDoc z poziomu wiersza poleceń działa tak samo — interakcja odbywa się w oknie uruchamiania.



Sztuczka na dziś dla PyDoc GUI: Jeżeli naciśniesz przycisk *open browser* w oknie przedstawionym na rysunku 15.3, PyDoc utworzy stronę indeksu zawierającą hiperłącza do każdego modułu, który możesz zaimportować. Obejmuje to standardowe moduły biblioteczne Pythona, moduły zainstalowanych rozszerzeń innych firm, moduły zdefiniowane przez użytkownika, znajdujące się na ścieżce wyszukiwania importów, a nawet moduły połączone statycznie lub dynamicznie z kodem napisanym w języku C. Bez PyDoc takie informacje trudno jest uzyskać bez napisania skryptu sprawdzającego wszystkie źródła modułów. W Pythonie 3.2 musisz to zrobić zaraz po uruchomieniu klienta GUI, ponieważ po rozpoczęciu wyszukiwania może to już nie działać w pełni. Powinieneś również pamiętać, że w PyDoc w trybie `-b` w wersji 3.2 i nowszych tę samą funkcjonalność masz na stronie początkowej, pokazanej na rysunku 15.1.

Nie tylko notki docstrings — pakiet Sphinx

Jeżeli szukasz sposobu na udokumentowanie swojego programu napisanego w języku Python w bardziej wyrafinowany sposób, możesz skorzystać z narzędzia o nazwie Sphinx (pakiet można pobrać ze strony <http://sphinx-doc.org>), z którego korzysta standardowa dokumentacja Pythona opisana w następnej sekcji oraz wiele innych projektów. Sphinx używa prostego języka znaczników *reStructuredText* i dziedziczy po pakiecie *Docutils* wiele narzędzi do analizy składniowej i tłumaczenia dokumentów napisanych w języku *reStructuredText*.

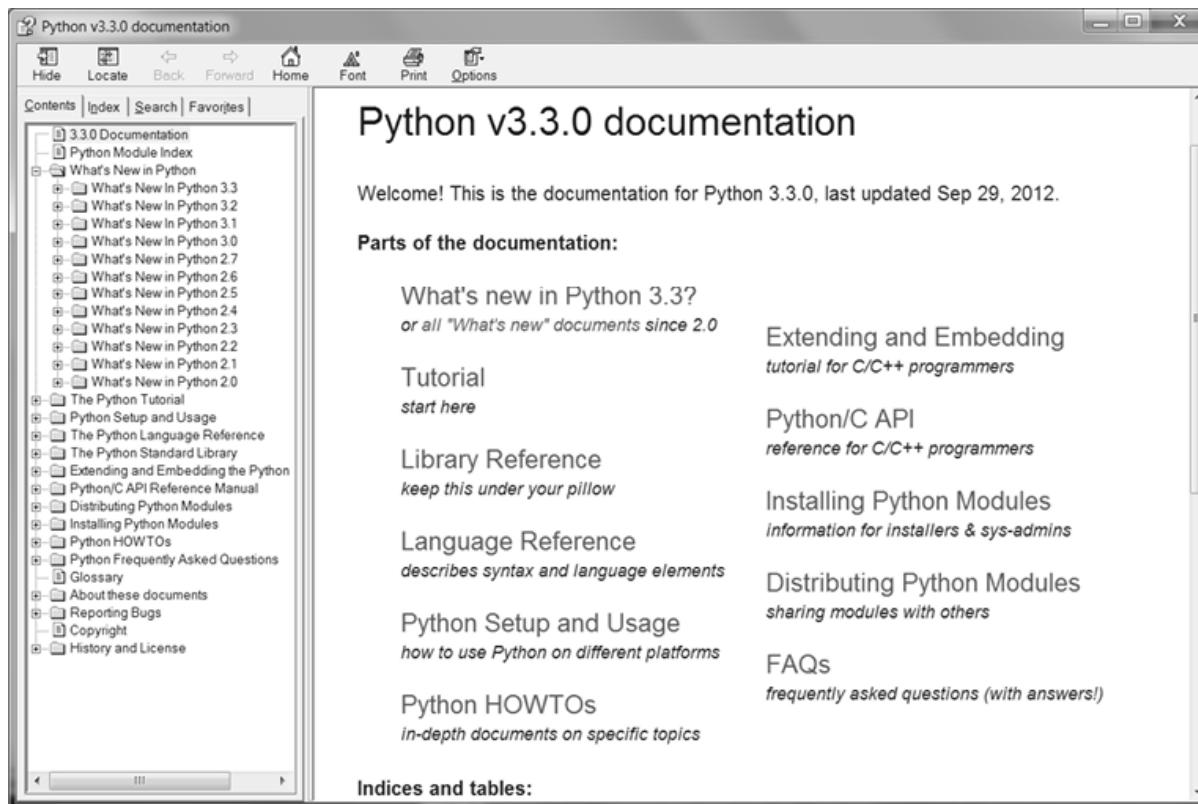
Sphinx obsługuje między innymi różne formaty wyjściowe (HTML, w tym Windows HTML Help, LaTeX dla wersji PDF do wydruku, strony podręcznika i zwykły tekst), obszerne i automatyczne odsyłacze, strukturę hierarchiczną z automatycznymi linkami do poszczególnych sekcji,

automatyczne indeksy, automatyczne podświetlanie kodu za pomocą pakietu *Pygments* (jest to samo w sobie godne uwagi narzędzie dla języka Python) i wiele innych. Oczywiście w przypadku mniejszych programów, w których dokumentacja i PyDoc mogą wystarczyć, korzystanie ze Sphinksa to trochę przesada, ale zastosowanie tego pakietu dla dużych projektów może z pewnością ułatwić tworzenie profesjonalnej dokumentacji. Więcej informacji na temat Sphinksa i powiązanych z nim narzędzi znajdziesz na stronie projektu i w innych źródłach w internecie.

Zbiór standardowej dokumentacji

Pełny, najbardziej aktualny opis języka oraz jego zbioru narzędzi można zawsze znaleźć w standardowej dokumentacji Pythona. Jest ona dostępna w formacie HTML i innych, a także instalowana w systemie Windows wraz z Pythonem — dostępna w menu *Python* przycisku *Start*, a także otwierana z menu *Help* wewnętrz IDLE. Dokumentację można również pobrać osobno ze strony internetowej <http://www.python.org> w różnych formatach lub przeglądać ją w internecie pod tym samym adresem (należy szukać odnośnika *Documentation*). W systemie Windows dokumentacja jest skompilowanym plikiem pomocy, który obsługuje wyszukiwanie, natomiast wersje dostępne w internecie na stronie *Python.org* zawierają wyszukiwarkę.

Po otwarciu dokumentacji w formacie dla systemu Windows wyświetlana jest strona główna podobna przedstawionej na rysunku 15.6. Dwa najważniejsze wpisy to najprawdopodobniej *Library Reference* (dokumentujący wbudowane typy, funkcje, wyjątki oraz moduły biblioteki standardowej), a także *Language Reference* (udostępniający formalny opis szczegółów na poziomie języka). Pod opcją *Tutorial* można znaleźć krótkie wprowadzenie przeznaczone dla początkujących użytkowników.



Rysunek 15.6. Zbiór standardowej dokumentacji Pythona dostępny na stronie <http://www.python.org>, z menu Help aplikacji IDLE oraz z odpowiedniego menu przycisku Start w systemie Windows. W systemie Windows to plik pomocy, który można przeszukiwać. Dla wersji publikowanych w internecie dostępna jest wyszukiwarka. Najczęściej używaną opcją jest Library Reference

Interesujące jest to, że dokumenty *What's New* (co nowego) w tym standardowym podręczniku zawierają kronikę zmian w Pythonie wprowadzanych w każdym wydaniu, począwszy od wersji 2.0, która pojawiła się pod koniec 2000 r. — to bardzo przydatne informacje dla osób przenoszących starszy kod w Pythonie do nowych wersji. Dokumenty te są niezastąpionym źródłem informacji na temat różnic w liniach językowych Pythona 2.x i 3.x, a także w ich standardowych bibliotekach.

Zasoby internetowe

Na oficjalnej stronie internetowej Pythona (<http://www.python.org>) można znaleźć odnośniki do różnych zasobów dotyczących Pythona; niektóre z nich dotyczą pewnych specjalnych tematów czy zastosowań. Po kliknięciu odnośnika *Documentation* można uzyskać dostęp do samouczka, a także przewodnika dla początkujących użytkowników (*Beginner's Guide*). Na stronie tej wymienione są również zasoby nieanglojęzyczne.

W internecie można dzisiaj znaleźć niezliczoną ilość stron wiki, blogów, witryn i najrozmaitszych innych źródeł poświęconych Pythonowi. Aby znaleźć społeczność programistów Pythona, wystarczy wyszukać w Google „programowanie w Pythonie” czy „Python programming”.

Publikowane książki

Można również sięgnąć po jedną z wielu książek dotyczących Pythona. Należy jednak pamiętać, że książki zawsze będą nieco do tyłu w stosunku do najświeższych zmian w tym języku ze względu na pracę włożoną w samo pisanie oraz naturalne opóźnienia wynikające z cyklu wydawniczego. Zazwyczaj kiedy książka zostaje już opublikowana, jest co najmniej trzy do czterech miesięcy w tyle za aktualnym stanem Pythona. W przeciwnieństwie do standardowej dokumentacji języka książki nie są również darmowe.

Dla wielu osób wygoda i jakość profesjonalnie publikowanego tekstu są warte tej ceny. Co więcej, Python zmienia się na tyle wolno, że książki poświęcone temu językowi zachowują ważność wiele lat po publikacji, w szczególności jeśli ich autorzy publikują uaktualnienia na swoich stronach internetowych. Wskazówki dotyczące innych książek poświęconych Pythonowi można znaleźć w „Przedmowie”.

Często spotykane problemy programistyczne

Przed przejściem do ćwiczeń do tej części książki przejrzymy jeszcze błędy popełniane często przy tworzeniu instrukcji i programów przez osoby początkujące. Wiele z nich odnosi się do ostrzeżeń, o których wspominałem już wcześniej w tej części książki — zostały one tutaj powtórzone w celu zebrania ich w jednym miejscu. W miarę nabierania doświadczenia w tworzeniu kodu w Pythonie każdy nauczy się unikać tych pułapek, jednak kilka dodatkowych uwag może już teraz pomóc w uniknięciu większych wpadek.

- **Nie należy zapominać o dwukropkach.** Zawsze należy pamiętać o umieszczeniu znaku `:` na końcu nagłówka instrukcji złożonej (pierwszym wierszu `if`, `while` czy `for`). Każdy o tym na początku zapomina (tak samo robiłem ja sam i większość z ponad trzech tysięcy moich studentów), jednak zawsze można się pocieszyć tym, że niedługo wejdzie nam to w nawyk.
- **Należy rozpoczynać kod w pierwszej kolumnie.** Trzeba pamiętać o zaczynaniu kodu najwyższego poziomu (niezagnieżdzonego) w kolumnie numer jeden. Dotyczy to niezagnieżdzonego kodu wpisywanego do plików modułów, ale także niezagnieżdzonego kodu wpisywanego w sesji interaktywnej.
- **Puste wiersze mają w sesji interaktywnej znaczenie.** Puste wiersze w instrukcjach złożonych są zawsze ignorowane w plikach modułów, ale kiedy kod wpisuje się w sesji interaktywnej, taki wiersz kończy instrukcję. Innymi słowy, puste wiersze mówią interaktywnemu wierszowi poleceń, że skończyliśmy instrukcję złożoną. Jeśli chcemy ją kontynuować, nie możemy nacisnąć przycisku `Enter` przy znaku zachęty `...` (lub w IDLE), dopóki naprawdę nie skończymy.
- **Należy stosować spójne wcięcia.** O ile nie wiemy, co edytor zrobi z tabulatorami, trzeba unikać mieszania tabulatorów i spacji we wcięciach bloku kodu. W przeciwnym razie może się okazać, że to, co widzimy w edytorze, może nie być tym samym, co zobaczy Python, kiedy tabulatory zamieni na pewną liczbę spacji. Tak samo jest w każdym innym języku ustrukturyzowanym blokowo, nie tylko w Pythonie. Jeżeli kolejny programista w inny sposób ustawi tabulatory, nie zrozumie struktury naszego kodu. Bezpieczniej jest używać w każdym bloku tylko tabulatorów lub tylko spacji.
- **Nie należy stosować w Pythonie stylu kodowania z języka C.** Przypomnienie dla programistów języków C i C++: wokół testów nagłówków `if` i `while` (na przykład `if (X == 1)`) nie trzeba umieszczać nawiasów. Można tak zrobić (każde wyrażenie można umieścić w nawiasach), jednak w tym kontekście są one całkowicie zbędne. Nie należy również kończyć wszystkich instrukcji średnikiem. Z technicznego punktu widzenia jest to w Pythonie dopuszczalne, jednak całkowicie bezużyteczne, o ile nie umieszczamy w jednym wierszu więcej niż jednej instrukcji (normalnie instrukcję kończy koniec wiersza). Należy też pamiętać, by nie osadzać instrukcji przypisania w testach pętli `while` i nie używać nawiasów klamrowych (`{}`) wokół bloków kodu (zamiast tego należy w spójny sposób je wcinać).
- **Zamiast while czy range należy używać prostych pętli for.** Kolejne przypomnienie: prosta pętla `for` (na przykład `for x in seq:`) jest prawie zawsze łatwiejsza do zapisania w kodzie i szybsza do wykonania od pętli licznika opartego na `while` czy `range`. Ponieważ Python wewnętrznie obsługuje indeksowanie dla prostego `for`, czasami pętla taka może być nawet dwa razy szybsza od odpowiadającej jej pętli `while`. Nie należy dać się skusić na ręczne odliczanie elementów w Pythonie!
- **Należy uważać na obiekty mutowalne w przypisaniach.** Wspomniałem o tym w rozdziale 11.: przypisując po kilka obiektów mutowalnych za jednym razem (na przykład `a = b = []`), a także korzystając z rozszerzonego przypisania (`a += [1, 2]`), należy bardzo uważać. W obu przypadkach modyfikacje w miejscu mogą wpływać na inne zmienne. Więcej informacji na ten temat można znaleźć w rozdziale 11.
- **Nie należy oczekiwac wyników od funkcji, które modyfikują obiekty w miejscu.** Z tym również spotkaliśmy się już wcześniej — operacje takie, jak metody `list.append` czy `list.sort`, wprowadzone w rozdziale 8., nie zwracają wartości (innych od `None`), dlatego powinno się je wywoływać bez przypisywania wyniku. Osoby poczynające nierzadko tworzą kod w stylu `mylist = mylist.append(X)`, by otrzymać wynik zastosowania metody `append`. Tak naprawdę jednak kod ten przypisuje listę `mylist` do `None`, a nie do zmodyfikowanej listy (i w rezultacie całkowicie tracimy referencję do listy).

Jeszcze bardziej podchwytliwym przykładem takiego zachowania jest w Pythonie 2.X próba przejścia posortowanych kluczy słownika. Często spotyka się kod w stylu `for k in D.keys().sort():`. Takie rozwiązanie prawie działa — metoda `keys` buduje listę kluczy, a metoda `sort` ją sortuje — jednak ponieważ metoda `sort` zwraca `None`, pętla nie powiedzie się, gdyż w rezultacie będzie pętlą po `None` (obiekcie niebędącym sekwencją). Taki kod nie działa tym bardziej w Pythonie 3.x, ponieważ klucze słowników są widokami, a nie listami!

Aby w poprawny sposób zapisać w kodzie to wyrażenie, należy albo skorzystać z nowszej funkcji `sorted` (zwracającej posortowaną listę), albo podzielić wywołania metody na instrukcje: `Ks = list(D.keys())`, potem `Ks.sort()`, a na końcu `for k in Ks:`. Jest to zresztą jeden z przypadków, w którym w pętli w sposób jawny będziemy chcieli wywołać metodę `keys`, zamiast polegać na iteratorach słowników — iteratory nie potrafią sortować.

- **W wywołaniach funkcji zawsze należy stosować nawiasy.** By wywołać funkcję, należy po jej nazwie umieścić nawiasy — bez względu na to, czy przyjmuje ona argumenty (trzeba zatem użyć kodu `funkcja()`, a nie samej nazwy `funkcja`). W czwartej części książki zobaczymy, że funkcje są po prostu obiektami, które mają specjalną operację — wywołanie wyzwalane za pomocą nawiasów. Można się do nich również odwoływać jak do każdego innego obiektu bez ich wywoływanego.

Na moich kursach problem ten zdaje się najczęściej występować z plikami. Wiele osób początkujących w celu zamknięcia pliku wpisuje `file.close`, a nie `file.close()`. Ponieważ odniesienie się do funkcji bez jej wywołania jest w Pythonie dopuszczalne, pierwsza, pozbawiona nawiasów wersja kończy się sukcesem, jednak nie zamyka pliku.

- **Nie należy używać rozszerzeń i ścieżek w operacjach importu i przeładowania.** W instrukcjach `import` należy pominąć ścieżki do katalogów oraz rozszerzenia plików (pisze się zatem `import mod`, a nie `import mod.py`). Podstawy modułów omówione zostały w rozdziale 3. i wróćmy do omawiania tej tematyki w piątej części książki. Ponieważ moduły mogą mieć rozszerzenia inne od `.py` (na przykład `.pyc`), zakodowanie danego rozszerzenia na stałe w kodzie jest nie tylko niepoprawne z punktu widzenia składni, ale dodatkowo nie ma większego sensu. Cała składnia ścieżek specyficzna dla danej platformy pochodzi z ustawień ścieżki wyszukiwania modułów, a nie z instrukcji `import`.
- **Inne pułapki.** Pamiętaj o ostrzeżeniami dotyczącymi wbudowanych typów na końcu poprzedniej części, ponieważ mogą one również kwalifikować się jako problemy z kodowaniem. Istnieją dodatkowe „pułapki”, które często pojawiają się w programowaniu w języku Python — utrata wbudowanej funkcji przez zmianę przypisania jej nazwy, ukrywanie modułu bibliotecznego przez użycie jego nazwy dla jednego ze swoich własnych, modyfikacja wartości domyślnych mutowalnych argumentów itd. — ale nie mamy jeszcze wystarczającej wiedzy, aby o nich teraz mówić. Aby dowiedzieć się więcej na temat tego, co powinieneś i czego nie powinieneś robić w Pythonie, musisz iść dalej; późniejsze części tej książki rozszerzają zestaw „pułapek”, który tutaj przedstawiliśmy.

Podsumowanie rozdziału

Niniejszy rozdział wprowadził nas w zagadnienia związane z dokumentacją kodu — zarówno piszącą przez nas samych dla tworzonych programów, jak i tą dostępną dla wbudowanych narzędzi. Omówiliśmy notki dokumentacyjne *docstrings*, a także zasoby dotyczące Pythona dostępne w internecie i na naszych komputerach. Zobaczyliśmy, jak funkcja `help` oraz interfejs strony internetowej mogą udostępnić dodatkowe źródła dokumentacji. Ponieważ jest to ostatni rozdział tej części książki, przyjrzelismy się również często popełnianym błędom z nadzieją uniknięcia ich w przyszłości.

W kolejnej części książki zaczniemy stosować znane nam już koncepcje do większej konstrukcji programu — funkcji. Zanim do tego dojdzie, powinieneś wykonać ćwiczenia dotyczące tej części książki umieszczone na końcu niniejszego rozdziału. A jeszcze wcześniej czas na wykonanie tradycyjnego quizu podsumowującego rozdział.

Sprawdź swoją wiedzę — quiz

1. Kiedy powinniśmy używać notek dokumentacyjnych zamiast komentarzy ze znakami #?
2. Podaj trzy sposoby przeglądania notek dokumentacyjnych.
3. W jaki sposób można uzyskać listę dostępnych atrybutów obiektu?
4. W jaki sposób można otrzymać listę wszystkich modułów dostępnych w Twoim systemie?
5. Jaką książkę o Pythonie powinieneś kupić po przeczytaniu tej, którą trzymasz w ręku?

Sprawdź swoją wiedzę — odpowiedzi

1. Notki dokumentacyjne (znane również jako notki *docstrings*) uznawane są za lepsze w przypadku większej, funkcjonalnej dokumentacji opisującej użycie modułów, funkcji, klas oraz metod w kodzie. Komentarze ze znakami # powinieneś stosować do opisywania bardziej zawiłych wyrażeń czy instrukcji. Taki podział wynika również z tego, że notki dokumentacyjne łatwiej jest znaleźć w pliku źródłowym, a dodatkowo po ekstrakcji można je wyświetlać za pośrednictwem systemu PyDoc.
2. Notki dokumentacyjne można zobaczyć, wyświetlając atrybut `__doc__` obiektu, przekazując obiekt do funkcji `help` pakietu PyDoc lub wybierając odpowiednie moduły z wyszukiwarki graficznego interfejsu użytkownika PyDoc opartego na języku HTML — albo w trybie `-g` klienta GUI w Pythonie 3.2 i wcześniejszych, albo w trybie `-b` dla wszystkich przeglądarek w Pythonie 3.2 i nowszych (wymagane od wersji 3.3). Oba rozwiązania działają w systemie klient-serwer, który wyświetla dokumentację w oknie przeglądarki internetowej. Dodatkowo PyDoc można wykorzystać do zapisania dokumentacji modułu w pliku HTML do późniejszego przeglądania czy drukowania.
3. Wbudowana funkcja `dir(X)` zwraca listę wszystkich atrybutów dołączonych do dowolnego obiektu. Lista składana w postaci wyrażenia `[a for a in dir(X) if not a.startswith('__')]` może być używana do odfiltrowania wewnętrznych nazw, które zaczynają się od podwójnych znaków podkreślenia (w kolejnej części książki pokażemy, jak można takie wyrażenie opakować w funkcję, aby ułatwić jej użycie).
4. W Pythonie 3.2 i wersjach wcześniejszych można uruchomić graficzny interfejs użytkownika pakietu PyDoc i wybrać opcję `open browser` (otwórz przeglądarkę); spowoduje to otwarcie strony internetowej zawierającej łącza do wszystkich modułów dostępnych dla Twoich programów. Ten tryb GUI nie działa już od wersji Python 3.3. W Pythonie 3.2 i nowszych wersjach tę samą funkcjonalność uzyskujemy, uruchamiając nowszy tryb dla wszystkich przeglądarek, używając w wierszu polecenia opcji `-b`; na pierwszej stronie wyświetlonej w przeglądarce w tym nowszym trybie pojawia się ten sam indeks z listą wszystkich dostępnych modułów.
5. Oczywiście moją... A tak naprawdę — o Pythonie napisano już setki książek; w „Przedmowie” znajdziesz listę kilku z nich, będących dobrym uzupełnieniem niniejszej; zarówno przewodników po samym języku, jak i książek opisujących jego praktyczne zastosowania.

Sprawdź swoją wiedzę — ćwiczenia do części trzeciej

Sprawdź swoją wiedzę — ćwiczenia do części

Skoro już wiemy, jak zapisywać w kodzie podstawową logikę programów, poniższe ćwiczenia będą od nas wymagały zaimplementowania określonych zadań za pomocą instrukcji. Najwięcej pracy będzie w ćwiczeniu 4., które pozwala nam sprawdzić alternatywne wersje kodu. Istnieje wiele sposobów układania instrukcji i część nauki Pythona polega na opanowaniu tego, które układy działają lepiej od innych. Po pewnym czasie w naturalny sposób będziesz skłaniał się ku temu, co doświadczeni programiści Python nazywają „najlepszymi praktykami”, ale najlepsze praktyki wymagają... praktyki.

Rozwiązania ćwiczeń znajdują się w podrozdziale „Część III — Instrukcje i składnia” w dodatku D.

1. Tworzenie podstawowych pętli.

- a. Napisz pętlę `for` wyświetlającą kod ASCII dla każdego znaku z łańcucha o nazwie `S`. Do konwersji każdego znaku na kod ASCII powinieneś wykorzystać wbudowaną funkcję `ord(znak)`. W celu sprawdzenia, jak ona działa, powinieneś ją przetestować w sesji interaktywnej.
- b. Następnie zmodyfikuj pętlę w taki sposób, by obliczała ona sumę kodów ASCII dla wszystkich znaków łańcucha.
- c. Na koniec wprowadź ostatnią modyfikację, tak by pętla zwracała listę zawierającą kody ASCII każdego znaku z łańcucha. Czy wyrażenie `map(ord, S)` ma podobny efekt? A co z wyrażeniem `[ord(c) for c in S]`? Dlaczego? (Wskazówka: patrz rozdział 14. książki).

2. Znaki lewego ukośnika.

Co się stanie, kiedy w sesji interaktywnej wpiszesz poniższy kod?

```
for i in range(50):
    print('Witaj %d\n\a' % i)
```

Pamiętaj, że poza środowiskiem IDLE ten przykład może spowodować piski komputera, zatem wykonywanie go w zatłoczonym pomieszczeniu niekoniecznie jest najlepszym pomysłem. IDLE, zamiast piszczeć, wyświetla na ekranie dziwne znaki (zobacz znaki ucieczki z tabeli 7.2).

3. *Sortowanie słowników.* W rozdziale 8. widzieliśmy, że słowniki są nieuporządkowanymi kolekcjami. Napisz pętlę `for` wyświetlającą elementy słownika w posortowanym (rosnącym) porządku. Wskazówka: użyj metod `keys` słowników i `sort` list lub nowszej funkcji wbudowanej `sorted`.
4. *Instrukcje warunkowe.* Przeanalizuj kod przedstawiony poniżej, wykorzystujący pętlę `while` oraz zmienną `found` do wyszukiwania na liście potęg liczby 2 wartości liczby 2 podniesionej do piątej potęgi (32). Kod ten zapisany jest w pliku modułu o nazwie `power.py`.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
found = False
```

```

i = 0
while not found and i < len(L):
    if 2 ** X == L[i]:
        found = True
    else:
        i = i+1
if found:
    print('pod indeksem', i)
else:
    print(X, 'nie odnaleziono')
C:\book\tests> python power.py
pod indeksem 5

```

W takiej postaci przykład ten nie wykorzystuje normalnych technik zapisu kodu w Pythonie. Postępuj zgodnie z poniższymi krokami w celu poprawienia kodu. W przypadku wszystkich transformacji możesz albo wpisać kod interaktywnie, albo zachować go w pliku wykonywanym z systemowego wiersza poleceń — skorzystanie z pliku bardzo ułatwia zadanie.

- Najpierw przepisz ten kod z użyciem części `else` pętli `while`, która pozwoli wyeliminować zmienną `found` oraz ostatnią instrukcję `if`.
- Później przepisz ten przykład w taki sposób, by wykorzystywał on pętlę `for` z częścią `else`, co pozwoli wyeliminować jawną logikę indeksującą listę. Wskazówka: by otrzymać indeks elementu, powinieneś skorzystać z metody listy `index` (`L.index(X)` zwraca wartość przesunięcia pierwszego `X` z listy `L`).
- Następnie całkowicie usuń pętlę `for`, przepisując przykład za pomocą prostego wyrażenia z operatorem przynależności `in`. Więcej informacji na ten temat możesz znaleźć w rozdziale 8.; możesz również przetestować to rozwiązanie, wpisując w sesji interaktywnej wyrażenie `2 in [1, 2, 3]`.
- Na koniec użyj pętli `for` oraz metody listy `append` w celu wygenerowania listy potęg liczby 2 o nazwie `L` w miejsce użycia zisanego na stałe w kodzie literała listy.

Kilka dodatkowych zagadnień:

- Czy użycie wyrażenia `2 ** X` poza pętlami poprawiło wydajność kodu? W jaki sposób można by zapisać takie rozwiązanie w kodzie?
 - Jak widzieliśmy ćwiczeniu 1., Python posiada narzędzie `map(funkcja, lista)`, które również może wygenerować listę potęg liczby 2 w następujący sposób: `map(lambda x: 2 ** x, range(7))`. Spróbuj wpisać ten kod w sesji interaktywnej. Z instrukcją `lambda` spotkamy się w kolejnej części książki, a w szczególności w rozdziale 19. Czy listy składane mogłyby nam tutaj pomóc (zobacz rozdział 14.)?
5. *Utrzymywanie kodu.* Jeżeli jeszczego nie zrobiłeś, poeksperymentuj z wprowadzaniem zmian w kodzie sugerowanych w ramce „Zmiana kolorów PyDoc”.

Duża część pracy związanej z tworzeniem prawdziwego oprogramowania polega na modyfikacjach istniejącego kodu, więc im szybciej zaczniesz to robić, tym lepiej. Dla porównania moja zmodyfikowana kopia pakietu PyDoc znajduje się w pakiecie przykładów książki, w pliku o nazwie *mypydoc.py*; aby zobaczyć, czym się różni, możesz uruchomić porównanie plików (polecenie `fc` w systemie Windows) z oryginalnym *pydoc.py* w wersji 3.3 (również dołączonej, ponieważ PyDoc w wersji 3.4 został radykalnie zmieniony, jak opisujemy w ramce). Jeżeli modyfikacja PyDoc zajmie Ci mniej czasu, niż przeczytanie tych słów, dostosuj kolory zgodnie z jego obecną konwencją; jeżeli będzie się to wiązało ze zmianą pliku CSS, to pozostaje nam mieć nadzieję, że cała procedura zostanie dobrze udokumentowana w podręcznikach Pythona.

[1] Warto zauważyc, że wyświetlanie pomocy dla obiektów typu łańcuch znaków (np. `help('')`) nie działa w najnowszych wersjach Pythona: zazwyczaj pomoc nie jest wyświetlana, ponieważ łańcuchy znaków są interpretowane w specjalny sposób — na przykład jako prośba o wyświetlenie pomocy dla niezimportowanego modułu (patrz wcześniej). W tym kontekście musisz użyć nazwy typu `str`, chociaż zarówno inne typy obiektów (`help([])`), jak i nazwy metod łańcuchów, do których odwołują się takie obiekty (`help('.Join')`), działają dobrze (przynajmniej tak było w Pythonie 3.3 — z czasem mogło to ulec zmianie). Istnieje również interaktywny tryb wyświetlania pomocy, który rozpoczynasz od wpisania polecenia `help()`.

Część IV Funkcje i generatory

Rozdział 16. Podstawy funkcji

W trzeciej części książki zapoznaliśmy się z instrukcjami proceduralnymi w Pythonie. Teraz przejdziemy do omówienia zbioru instrukcji i wyrażeń, których można użyć do tworzenia własnych funkcji.

Mówiąc w uproszczeniu, *funkcja* jest komponentem grupującym zbiór instrukcji w taki sposób, by mogły one być wykonane w programie więcej niż jeden raz — inaczej mówiąc, to rodzaj procedury wywoywanej po nazwie. Funkcje obliczają i zwracają wynik swojego działania oraz pozwalają określać parametry służące za dane wejściowe — mogą się one zatem zmieniać z każdym wykonaniem kodu. Zapisanie zbioru działań w postaci funkcji sprawia, że staje się ona przydatnym narzędziem, z którego można korzystać w wielu różnych kontekstach.

Co jednak ważniejsze, funkcje są alternatywą dla programowania polegającego *na kopiowaniu i wklejaniu fragmentów kodu*. Zamiast tworzyć wiele nadmiarowych kopii tego samego kodu, możemy wykorzystać jedną funkcję. W ten sposób radykalnie ograniczamy ilość pracy w przyszłości — jeżeli operacja będzie musiała kiedyś zostać zmodyfikowana, niezbędne będzie dokonanie zmian tylko w jednej funkcji, a nie w wielu fragmentach kodu rozproszonych w całym programie.

Funkcje są również najbardziej podstawową strukturą programu, która maksymalizuje możliwości ponownego użycia kodu i prowadzi do szerszych zagadnień dotyczących projektowania programów. Jak się niebawem przekonasz, funkcje pozwalają na dzielenie złożonych systemów na mniejsze, łatwiejsze do zarządzania części. Wdrażając każdą z części jako funkcję, sprawiamy, że jest ona wielokrotnego użytku i łatwiejsza do kodowania.

W tabeli 16.1 przedstawiono podstawowe narzędzia związane z funkcjami, które omówimy w tej części książki — jest to zestaw zawierający wywołania funkcji, dwa sposoby tworzenia funkcji (`def` i `lambda`), dwa sposoby zarządzania zasięgiem funkcji (`global` i `nonlocal`) oraz dwa sposoby zwracania wyników z powrotem do miejsca wywołania funkcji (`return` i `yield`).

Tabela 16.1. Instrukcje oraz wyrażenia powiązane z funkcjami

Instrukcja lub wyrażenie	Przykłady
Wywołania	<code>myfunc('mielonka', 'jajka', meat=ham, *rest)</code>
<code>def</code>	<code>def printer(message):</code> <code>print('Witaj ' + message)</code>
<code>return</code>	<code>def adder(a, b=1, *c):</code> <code>return a + b + c[0]</code>
<code>global</code>	<code>x = 'stary'</code> <code>def changer():</code> <code>global x; x = 'nowy'</code>
<code>nonlocal (3.x)</code>	<code>def outer():</code> <code>x = 'stary'</code> <code>def changer():</code> <code>global x; x = 'nowy'</code>

yield	<pre>def squares(x): for i in range(x): yield i ** 2</pre>
lambda	<pre>funcs = [lambda x: x**2, lambda x: x**3]</pre>

Dlaczego używamy funkcji

Zanim przejdziemy do szczegółów, warto zarysować pełny obraz tego, czym tak naprawdę są funkcje. Funkcje są prawie uniwersalnym narzędziem nadającym strukturę programowi. Można je spotkać w innych językach programowania, w których czasami nazywane są *procedurami* czy *podprocedurami*. Uogólniając: funkcje w programie spełniają dwie podstawowe role.

Maksymalizacja ponownego wykorzystania kodu i minimalizacja jego powtarzalności

Tak jak w większości języków programowania, funkcje Pythona są najprostszym sposobem spakowania razem logiki, z której możemy chcieć korzystać w więcej niż jednym miejscu i częściej niż raz. Dotychczas cały wpisywany kod wykonywany był natychmiast. Funkcje pozwalają na grupowanie i uogólnienie kodu, tak by mógł on później być wykorzystany w dowolny sposób i wiele razy. Ponieważ pozwalają na zapisanie operacji w jednym miejscu i wykorzystanie jej w wielu miejscach, funkcje Pythona są podstawowym narzędziem *faktoryzacji* dostępnym w tym języku. Pozwalają na zredukowanie powtarzalności kodu w programach i tym samym zmniejszają wysiłek związany z utrzymaniem kodu w przyszłości.

Proceduralne podzielenie na części

Funkcje umożliwiają również dzielenie systemów na fragmenty z jasno zdefiniowanymi rolami. By na przykład zrobić pizzę od podstaw, zaczynamy od zagniecenia ciasta, nałożenia go na blachę, ułożenia dodatków, pieczenia i tak dalej. Gdybyśmy programowali robota robiącego pizzę, funkcje pomogłyby nam podzielić całe zadanie „zrób pizzę” na kilka części — po jednej funkcji dla każdego podzadania tego procesu. Łatwiej jest implementować mniejsze zadania w izolacji niż cały proces naraz. Funkcje wiążą się z *procedurami* — sposobami zrobienia czegoś, a nie celem tego działania. W szóstej części książki, gdy zaczniemy tworzyć nowe obiekty za pomocą klas, zobaczymy, dlaczego to rozróżnienie ma takie znaczenie.

W tej części książki omówimy narzędzia wykorzystywane do tworzenia funkcji w Pythonie — podstawy funkcji, reguły dotyczące zasięgu zmiennych, przekazywanie argumentów, a także kilka powiązanych z nimi zagadnień, takich jak generatory oraz narzędzia funkcyjne. Ponieważ jego znaczenie zaczyna być coraz bardziej widoczne na tym poziomie kodowania, powrócimy również do pojęcia polimorfizmu, które zostało wprowadzone wcześniej w książce. Jak zobaczymy niebawem, funkcje nie wnoszą wiele nowej składni, jednak prowadzą nas do większych koncepcji programistycznych.

Tworzenie funkcji

Choć raczej w sposób nieoficjalny, używaliśmy już funkcji we wcześniejszych rozdziałach. By na przykład utworzyć obiekt pliku, wywoływałyśmy wbudowaną funkcję `open`. W podobny sposób użyliśmy funkcji wbudowanej `len` do obliczenia liczby elementów obiektu kolekcji.

W niniejszym rozdziale dowiemy się, jak w Pythonie pisze się *nowe* funkcje. Funkcje pisane przez nas zachowują się w ten sam sposób jak prezentowane już funkcje wbudowane — są wywoływane w wyrażeniach, przekazuje się do nich wartości i zwraca się z nich wyniki. Pisanie

nowych funkcji wymaga jednak zastosowania kilku nowych koncepcji, o których jeszcze nie wspominaliśmy. Co więcej, w Pythonie funkcje zachowują się bardzo różnie od funkcji z języków kompilowanych, takich jak C. Poniżej znajduje się krótkie wprowadzenie do najważniejszych koncepcji dotyczących funkcji w Pythonie — wszystkie te kwestie omówimy w tej części książki.

- **def to kod wykonywalny.** Funkcje w Pythonie są definiowane za pomocą nowej instrukcji `def`. W przeciwieństwie do funkcji z języków kompilowanych (takich, jak C) `def` jest instrukcją wykonywalną — funkcja w Pythonie nie istnieje, dopóki Python nie dotrze do jej kodu i nie wykona instrukcji `def`. Tak naprawdę poprawne (i nawet czasami użyteczne) jest zagnieźdzanie instrukcji `def` wewnątrz instrukcji `if`, pętli `while`, a nawet innych instrukcji `def`. W typowej operacji instrukcje `def` zapisywane są w plikach modułów i wykonywane w celu wygenerowania funkcji, kiedy plik modułu zostaje zimportowany po raz pierwszy.
- **Instrukcja def tworzy obiekt i przypisuje go do nazwy.** Kiedy Python dotrze do instrukcji `def` i ją wykona, generowany jest nowy obiekt funkcji, który zostaje przypisany do nazwy funkcji. Tak jak w przypadku wszystkich operacji przypisania, nazwa funkcji staje się referencją do obiektu funkcji. W nazwie funkcji nie ma nic magicznego — jak zobaczymy, obiekt funkcji można przypisać do innych nazw czy przechować w liście. Do obiektów funkcji można także dołączać dowolne zdefiniowane przez użytkownika *atrybuty* w celu przechowywania danych.
- **Wyrażenie lambda tworzy obiekt i zwraca go jako wynik.** Funkcje można również tworzyć za pomocą wyrażenia `lambda` — opcji pozwalającej na wykorzystywanie definicji funkcji wewnątrz wiersza w miejscach, w których składnia instrukcji `def` nie będzie działała (jest to bardziej skomplikowana koncepcja omówiona w rozdziale 19.).
- **Instrukcja return przesyła obiekt wynikowy z powrotem do obiektu wywołującego.** Kiedy wywołuje się funkcję, działanie kod wywołującego zatrzymuje się do czasu, aż funkcja zakończy pracę i zwróci do niego sterowanie. Funkcje obliczające wartość przesyłają ją z powrotem do wywołującego za pomocą instrukcji `return`, a zwieracana wartość staje się wynikiem wywołania funkcji. Wykonanie instrukcji `return` bez żadnej wartości zwraca po prostu sterowanie do obiektu wywołującego i przesyła do niego rezultat domyślny, czyli obiekt `None`.
- **Instrukcja yield również odsyła obiekt wynikowy z powrotem do obiektu wywołującego, ale zapamiętuje, gdzie zakończyła działanie.** Funkcje znane jako *generatory* mogą również wykorzystywać instrukcję `yield` do odsyłania wartości i zawieszenia stanu w taki sposób, by można je było kontynuować później w celu otrzymania serii wyników rozłożonych w czasie. To kolejne zaawansowane zagadnienie, które zostanie omówione później w tej części książki.
- **Instrukcja global pozwala na deklarowanie zmiennych, które będą przypisywane na poziomie modułu.** Domyślnie wszystkie nazwy przypisane w funkcji są lokalne dla tej funkcji i istnieją tylko na czas jej wykonywania. Aby przypisać jakąś nazwę w module, w którym zdefiniowana jest funkcja, taka nazwa musi zostać zadeklarowana przy użyciu instrukcji `global`. Ogólnie rzecz biorąc, nazwy zawsze są wyszukiwane w obrębie ich zasięgów (ang. *scopes*) czyli obszarów, w których są dostępne; przypisanie wiąże nazwę z zasięgiem.
- **Instrukcja nonlocal pozwala na deklarowanie zmiennych z funkcji otaczającej, które mają być przypisane.** Podobnie jak instrukcja `global`, dodana w Pythonie 3.x instrukcja `nonlocal` pozwala funkcji na przypisanie zmiennej istniejącej w zasięgu instrukcji `def` jej funkcji otaczającej. Pozwala to na wykorzystywanie funkcji zawierających do przechowywania *stanu* — informacji zapamiętanej między kolejnymi wywołaniami funkcji — bez konieczności korzystania ze współdzielonych zmiennych globalnych.
- **Argumenty przekazywane są przez przypisanie (referencję obiektu).** W Pythonie argumenty przekazywane są do funkcji za pomocą przypisania (co, jak już wiemy, oznacza referencję do obiektu). Jak zobaczymy niebawem, w modelu Pythona kod wywołujący i funkcja współdzielą obiekty przez referencje, jednak nie ma tutaj aliasów nazw. Zmiana nazwy argumentu wewnętrz funkcji nie zmienia odpowiadającej jej nazwy w kodzie

wywołującym, ale zmiana przekazanych obiektów mutowalnych w miejscu może zmienić obiekty współdzielone przez wywołującego i służyć jako wynik funkcji.

- **Argumenty są domyślnie przekazywane według pozycji, chyba że określmy inaczej.** Wartości przekazywane w wywołaniu funkcji domyślnie odpowiadają nazwom argumentów w definicji funkcji od lewej do prawej. Dla zachowania elastyczności w wywołaniach funkcji możemy również przekazywać argumenty, używając par `nazwa=wartość`, lub rozpakowywać dowolną liczbę argumentów i przekazywać je z użyciem wyrażeń `*pargs` i `**kwargs`. Definicje funkcji używają tych samych dwóch form do określania wartości domyślnych argumentów i gromadzenia dowolnej liczby otrzymanych argumentów.
- **Argumenty, zwarcane wartości i zmienne nie są deklarowane.** Tak jak w każdym innym przypadku w Pythonie, w funkcjach nie ma ograniczenia w zakresie typów. Niczego dotyczącego funkcji nie trzeba tak naprawdę wcześniej deklarować — można zatem przekazać argumenty dowolnego typu czy zwrócić dowolny rodzaj obiektu. W rezultacie jedną funkcję można zastosować do różnych typów obiektów — każdy obiekt zawierający zgodny *interfejs* (metody i wyrażenia) będzie dobry, bez względu na typ.

Jeżeli cokolwiek z powyższego opisu nie jest zrozumiałe — nie masz się co martwić. Wszystkie przedstawione koncepcje zostaną zilustrowane w tej części książki praktycznymi przykładami kodu. Zaczniemy od rozszerzenia niektórych z powyższych zagadnień i przyjrzenia się kilku przykładom.

Instrukcje def

Instrukcja `def` tworzy obiekt funkcji i przypisuje go do nazwy. Jej ogólny format jest następujący:

```
def nazwa(arg1, arg2,... argN):  
    instrukcje
```

Tak jak w przypadku wszystkich instrukcji złożonych Pythona, `def` składa się z wiersza nagłówka, po którym następuje zazwyczaj wcięty blok instrukcji (lub prosta instrukcja umieszczona po dwukropku). Blok instrukcji staje się *ciałem* funkcji, czyli kodem wykonywanym przez Pythona, za każdym razem, gdy funkcja jest wywoływana.

Wiersz nagłówka z `def` określa *nazwę* funkcji przypisywaną do obiektu funkcji, a także zero lub większą liczbę *argumentów* (czasami nazywanych *parametrami*) znajdujących się w nawiasach. Nazwy argumentów w nagłówku przypisywane są do obiektów przekazanych w nawiasach w momencie wywołania funkcji.

Ciało funkcji często zawiera instrukcję `return`.

```
def nazwa(arg1, arg2,... argN):  
    ...  
    return wartość
```

Instrukcja `return` Pythona może pojawić się w dowolnym miejscu ciała funkcji. Kiedy zostanie napotkana, kończy działanie funkcji i odsyła wyniki z powrotem do miejsca wywołującego. Instrukcja `return` może zawierać wyrażenie podające wynik działania funkcji. Jeżeli taki wynik zostanie pominięty, instrukcja `return` domyślnie zwraca obiekt `None`.

Instrukcja `return` sama w sobie jest opcjonalna — jeżeli nie występuje w kodzie, funkcja kończy działanie, kiedy sterowanie wychodzi poza jej ciało. Z technicznego punktu widzenia funkcja bez instrukcji `return` automatycznie zwraca obiekt `None`, jednak w takiej sytuacji zwracana wartość jest zazwyczaj ignorowana w miejscu wywołania.

Funkcje mogą również zawierać instrukcję `yield`, zaprojektowane w taki sposób, aby generować szeregi wartości w czasie, jednak ich omówienie odłożymy do czasu przedstawienia w rozdziale 20. zagadnień związanych z generatorami.

Instrukcja `def` uruchamiana jest w czasie wykonania

Instrukcja `def` Pythona jest prawdziwą instrukcją wykonywalną — kiedy jest wykonywana, tworzy nowy obiekt funkcji i przypisuje go do nazwy (należy pamiętać, że w Pythonie istnieje tylko *czas wykonywania* — nie ma osobnego czasu komplikacji). Ponieważ jest instrukcją, może pojawiać się w każdym miejscu, w jakim występują instrukcje — nawet zagnieżdżona w innych instrukcjach. Choć instrukcje `def` są zazwyczaj wykonywane, kiedy importowany jest zawierający je moduł, można je również umieścić w instrukcji `if` pozwalającej na wybór alternatywnej definicji funkcji.

```
if test:  
    def func():                      # Zdefiniowanie funkcji w taki  
    sposób  
    ...  
else:  
    def func():                      # Albo w taki sposób  
    ...  
    ...  
func()                                # Wywołanie wybranej i zbudowanej  
wersji
```

Jedną z metod zrozumienia tego kodu jest uświadomienie sobie, że `def` przypomina instrukcję `=`, ponieważ po prostu przypisuje nazwę w czasie wykonywania. W przeciwieństwie do języków kompilowanych, takich jak C, funkcje Pythona nie muszą być w pełni zdefiniowane przed wykonaniem programu. Instrukcje `def` nie są analizowane, dopóki nie zostaną wykonane, a kod *wewnętrzny* `def` nie zostanie wykonany aż do momentu, w którym funkcja ta zostanie później wywołana.

Ponieważ definiowanie funkcji odbywa się w czasie jej wykonywania, w samej nazwie funkcji nie ma nic specjalnego. Ważny jest obiekt, do którego nazwa ta się odnosi.

```
othername = func                      # Przypisanie obiektu funkcji  
othername()                            # Ponowne wywołanie funkcji func
```

W powyższym kodzie funkcja `func` została przypisana do innej nazwy i wywołana już za jej pomocą. Tak jak wszystko inne w Pythonie, funkcje są po prostu *obiektami*. Są zapisywane w jawnym sposób w pamięci w momencie wykonywania programu. Tak naprawdę poza wywoływaniem funkcje pozwalają na dołączanie dowolnych *atrybutów* w celu zapisania informacji i późniejszego ich wykorzystania:

```
def func(): ...                        # Utworzenie obiektu funkcji  
func()                                 # Wywołanie obiektu  
func.attr = value                      # Dołączenie atrybutów
```

Pierwszy przykład – definicje i wywoływanie

Poza powyższymi kwestiami związanymi z czasem wykonywania (które dla programistów pracujących z tradycyjnymi językami kompilowanymi wydają się dość wyjątkowe) funkcje Pythona są stosunkowo proste w użyciu. Przejdzmy zatem do pierwszego prawdziwego przykładu, który pozwoli nam zademonstrować podstawy. Jak wynika z powyższego opisu, na funkcję składają się dwa elementy — *definicja* (instrukcja `def` tworząca funkcję) oraz *wywołanie* (wyrażenie nakazujące Pythonowi wykonanie ciała funkcji).

Definicja

Poniżej znajduje się definicja wpisana w sesji interaktywnej, definiująca funkcję o nazwie `times`, która zwraca iloczyn dwóch argumentów.

```
>>> def times(x, y):          # Utworzenie i przypisanie
    funkcji
        ... return x * y       # Ciało funkcji wykonywane po
    wywołaniu
    ...
```

Kiedy Python trafi w kodzie na `def` i wykona tę instrukcję, utworzy nowy obiekt funkcji zawierający jej kod i przypisze ten obiekt do nazwy `times`. Zazwyczaj taka instrukcja znajduje się w pliku modułu i wykonywana jest, kiedy importowany jest zawierający ją plik. W przypadku tak niewielkiej funkcji sesja interaktywna będzie jednak w zupełności wystarczająca.

Wywołanie

Instrukcja `def` definiuje funkcję, ale jej nie wywołuje. Po wykonaniu instrukcji `def` możemy wywołać (wykonać) funkcję w programie, dodając nawiasy po jej nazwie. Nawiasy mogą opcjonalnie zawierać jeden lub większą liczbę argumentów, które mają być przekazane (przypisane) do zmiennych z nagłówka funkcji.

```
>>> times(2, 4)           # Argumenty w nawiasach
8
```

Powyższe wyrażenie przekazuje do funkcji `times` dwa argumenty. Jak wspomniano wcześniej, argumenty przekazywane są przez przypisanie, dlatego w tym przypadku zmienna `x` z nagłówka funkcji przypisywana jest do wartości 2, natomiast zmienna `y` przypisywana jest do wartości 4. Później wykonywane jest ciało funkcji. W tej funkcji ciało składa się z jednej instrukcji `return` odsyłającej wynik w postaci wartości wyrażenia wywołującego. Zwracany obiekt został tutaj wyświetlony w sesji interaktywnej (tak, jak w większości języków, iloczyn $2 * 4$ wynosi w Pythonie 8), jednak gdybyśmy potrzebowali użyć go później, moglibyśmy go przypisać do zmiennej — jak w poniższym kodzie.

```
>>> x = times(3.14, 4)      # Zapisanie obiektu wyniku
>>> x
12.56
```

Sprawdźmy teraz, co się stanie, kiedy funkcja zostanie wywołana trzeci raz, z zupełnie innymi obiekttami przekazanymi w charakterze argumentów.

```
>>> times('Ni', 4) # Dla funkcji typ nie ma
znaczenia
'NiNiNiNi'
```

Tym razem nasza funkcja oznacza coś zupełnie innego (i zawiera kolejne odniesienie do Monty Pythona). W trzecim wywołaniu do zmiennych `x` oraz `y` w miejsce dwóch liczb przekazywane są łańcuch znaków i liczba całkowita. Jak wiemy, operator `*` działa zarówno na liczbach, jak i sekwencjach. Ponieważ w Pythonie nigdy nie deklarujemy typów zmiennych, argumentów ani zwracanych wartości, możemy wykorzystać funkcję `times` zarówno do *pomnożenia* liczb, jak i *powtórzenia* sekwencji.

Innymi słowy, to, co oznacza i robi nasza funkcja `times`, zależy od tego, co do niej przekażemy. Jest to w Pythonie kwestia kluczowa (i tym samym chyba klucz do wykorzystywania całego języka), która zasługuje na nieco bardziej szczegółowe wyjaśnienie.

Polimorfizm w Pythonie

Jak widzieliśmy przed momentem, znaczenie wyrażenia `x * y` w naszej prostej funkcji `times` jest w całości uzależnione od typów obiektów, jakimi są `x` oraz `y`. Tym samym jedna funkcja może w jednym przypadku wykonywać mnożenie, a w drugim powtarzanie. Python pozostawia *obiektem* zrobienie czegoś, co ma sens w przypadku podanej składni. Tak naprawdę znak `*` jest po prostu mechanizmem przekazującym kontrolę do przetwarzanych obiektów.

Taki rodzaj zachowania uzależnionego od typów znany jest pod nazwą *polimorfizmu* — koncepcji, z którą pierwszy raz spotkaliśmy się w rozdziale 4. i która oznacza, że operacja uzależniona jest od obiektów, na jakich jest wykonywana. Ponieważ Python jest językiem o typach dynamicznych, szerzy się w nim polimorfizm. Tak naprawdę w Pythonie *każda* operacja jest polimorficzna — wyświetlanie, indeksowanie, operator `*` i wiele, wiele innych.

Jest to celowe i z takiego rozwiązania wynika duża część związkowości oraz elastyczności Pythona. Pojedyncza funkcja może na przykład zostać automatycznie zastosowana do szerokiej gamy typów obiektów. Dopóki obiekty te obsługują oczekiwany interfejs (inaczej mówiąc: protokół), funkcja może je przetworzyć. Oznacza to, że jeśli obiekty przekazane do funkcji mają oczekiwane metody oraz operatory wyrażeń, są z miejsca zgodne z logiką funkcji.

Nawet w naszej prostej funkcji `times` oznaczało to, że *dowolne* dwa obiekty obsługujące operator `*` będą działać — bez względu na to, czym są, i na to, kiedy zostaną zapisane w kodzie. Funkcja ta będzie działała na dwóch liczbach (wykonując mnożenie) lub łańcuchu znaków i liczbie (wykonując powtórzenie) bądź dowolnej kombinacji obiektów obsługujących oczekiwany interfejs — nawet obiektów opartych na klasach, których kod jeszcze nie powstał.

Co więcej, jeśli przekazane obiekty *nie* obsługują oczekiwanej interfejsu, Python wykryje błąd przy wykonywaniu wyrażenia z operatorem `*` i automatycznie zwróci wyjątek. Samodzielne sprawdzanie błędów w kodzie jest zatem bezcelowe. Tak naprawdę takie działanie obniżyłoby użyteczność funkcji, ponieważ byłaby ona ograniczona do obiektów, na których jej działanie zostało przetestowane.

Okazuje się to kluczową różnicą w filozofii Pythona i języków z typami statycznymi, takich jak C++ i Java. W Pythonie określone typy danych *mają nie mieć znaczenia* dla kodu. Jeśli tak nie jest, kod będzie ograniczony do działania wyłącznie na typach przewidzianych w momencie pisania go i nie będzie obsługiwał innych zgodnych typów obiektów, które mogą powstać w przyszłości. Choć można testować kod pod kątem działania na określonych typach za pomocą wbudowanych narzędzi, takich jak funkcja `type`, takie działanie niszczy elastyczność kodu. W Pythonie tworzy się kod z myślą o *interfejsach* obiektów, a nie typach danych^[1].

Oczywiście niektóre programy mają bardzo specyficzne wymagania i taki polimorficzny model programowania oznacza, że zamiast stosować deklaracje typów, których kompilator mógłby używać do automatycznego wykrywania pewnych rodzajów błędów, musimy samodzielnie

sprawdzać tworzony kod. Jednak w zamian za nieco więcej początkowego testowania możemy radykalnie zmniejszyć ilość kodu, jaką musimy napisać, a także radykalnie zwiększyć jego elastyczność. Jak się niebawem okaże, w praktyce oznacza to znaczący zysk netto.

Drugi przykład – przecinające się sekwencje

Przyjrzyjmy się kolejnemu przykładowi, który robi coś nieco bardziej przydatnego od mnożenia argumentów i dodatkowo ilustruje podstawy funkcji.

W rozdziale 13. utworzyliśmy pętlę `for` zbierającą elementy wspólne dla dwóch łańcuchów znaków. Zauważliśmy tam, że kod nie był tak użyteczny, jak mógłby być, ponieważ został skonfigurowany do działania jedynie na określonych zmiennych i nie mógł być wykonany ponownie później. Oczywiście moglibyśmy skopiować kod i wkleić go do każdego miejsca, w którym miałyby ona zostać wykonane, jednak takie rozwiązanie nie jest ani dobre, ani uniwersalne — nadal musielibyśmy dokonywać edycji każdej kopii kodu w celu obsłużenia różnych nazw sekwencji. Zmiana samego algorytmu wymagałaby wprowadzenia modyfikacji do każdej z kopii.

Definicja

W tej chwili wszyscy już zapewne się domyślają, że rozwiązaniem tego problemu jest umieszczenie pętli `for` wewnętrz funkcji. Takie działanie ma kilka zalet.

- Umieszczenie kodu w funkcji sprawia, że uzyskujemy narzędzie, które można wykonać tyle razy, ile nam się będzie podobało.
- Ponieważ kod wywołujący może przekazywać argumenty dowolnego typu, funkcje są na tyle uniwersalne, by działały na dowolnych dwóch sekwencjach (czy innych obiektach, na których można wykonywać iterację), jakich część wspólną chcemy obliczyć.
- Kiedy logika umieszczana jest w funkcji, w momencie gdy chcemy zmienić jej sposób działania, wystarczy jedynie wprowadzić modyfikacje w jednym miejscu.
- Umieszczenie funkcji w pliku modułu oznacza, że można ją importować i użyć jej ponownie w dowolnym programie działającym na naszym komputerze.

W rezultacie opakowanie kodu w funkcję sprawia, że staje się ona wszechstronnym narzędziem do zwracania części wspólnych.

```
def intersect(seq1, seq2):  
    res = []                                # Na początek pusta lista  
    for x in seq1:                          # Przeszukanie pierwszej  
        sekwencji  
            if x in seq2:                    # Powtarzający się element?  
                res.append(x)               # Dodanie na końcu listy wyników  
    return res
```

Transformacja z prostego kodu z rozdziału 13. na funkcję jest prosta — po prostu osadziliśmy oryginalny kod pod nagłówkiem `def`, a z obiektów, na których działa funkcja, zrobiliśmy przekazywane nazwy argumentów. Ponieważ funkcja ta oblicza wynik, dodaliśmy również instrukcję `return` przesyającą ten wynik z powrotem do kodu wywołującego.

Wywołania

Zanim wywołamy funkcję, musimy ją utworzyć. Aby to zrobić, należy wykonać jej instrukcję `def` — albo w sesji interaktywnej, albo umieszczając ją w pliku modułu i importując ten plik. Po wykonaniu instrukcji `def` można wywołać funkcję, przekazując jej dowolne dwa obiekty sekwencji w nawiasach.

```
>>> s1 = "mielonka"
>>> s2 = "biedronka"
>>> intersect(s1, s2)                                # Łańcuchy znaków
['i', 'e', 'o', 'n', 'k', 'a']
```

W powyższym kodzie przekazaliśmy do funkcji dwa łańcuchy znaków i otrzymaliśmy z powrotem listę zawierającą znaki wspólne w obu łańcuchach. Algorytm wykorzystywany przez funkcję jest prosty: „Dla każdego elementu z pierwszego argumentu, jeżeli dany element znajduje się również w drugim argumentie, dodaj go do wyniku”. W Pythonie jest to nieco krótsze niż w języku polskim, jednak działa tak samo.

Uczciwie należy przyznać, że nasza funkcja jest dość wolna (wykonuje zagnieżdżone pętle), nie zwraca tak naprawdę matematycznej części wspólnej (w wyniku mogą się pojawiać duplikaty), a na dodatek jest zupełnie niepotrzebna (widzieliśmy już, że typ danych zbioru w Pythonie udostępnia wbudowaną operację obliczania części wspólnej). I faktycznie, funkcję tę można zastąpić prostym wyrażeniem listy składanej, gdyż działa ono zgodnie z klasycznym wzorcem pętli zbierającej elementy:

```
>>> [x for x in s1 if x in s2]
['i', 'e', 'o', 'n', 'k', 'a']
```

Kod ten spełnia jednak swoje zadanie prostego przykładu działania funkcji. Ten fragment kodu można zastosować do różnych typów obiektów, co zostanie wyjaśnione w kolejnym podrozdziale. W rozdziale 18. poprawimy i rozszerzymy ten przykład tak, aby obsługiwać dowolnie wiele operandów, ale najpierw musimy się dowiedzieć więcej o trybach przekazywania argumentów.

Raz jeszcze o polimorfizmie

Tak jak wszystkie dobre funkcje w Pythonie, funkcja `intersect` jest polimorficzna. Oznacza to, że działa na dowolnych typach, które obsługują oczekiwany interfejs obiektu.

```
>>> x = intersect([1, 2, 3], (1, 4))      # Typy mieszane
>>> x                                     # Zapisany obiekt wyniku
[1]
```

Tym razem do funkcji przekazaliśmy różne typy obiektów — listę oraz krotkę (typy mieszane); funkcja nadal była w stanie wybrać elementy wspólne. Ponieważ nie musimy określać z wyprzedzeniem typów argumentów, funkcja `intersect` z radością przejdzie dowolny typ sekwencji, o ile tylko obsługuje on oczekiwany interfejs.

Dla `intersect` oznacza to, że pierwszy argument musi obsługiwać pętlę `for`, a drugi test przynależności `in`. Funkcja będzie działała na dwóch dowolnych obiektach tego rodzaju bez względu na ich typ — obejmuje to zarówno sekwencje przechowywane fizycznie w pamięci, jak i łańcuchy znaków oraz listy, wszystkie typy z rozdziału 14., na których można wykonywać iterację, w tym pliki i słowniki, a nawet utworzone przez nas obiekty oparte na klasach

wykorzystujące techniki przeciążania operatorów (omówimy je później, w dalszej części książki) [2].

I znów, jeśli przekażemy obiekty nieobsługujące tych interfejsów (na przykład liczby), Python automatycznie wykryje niedopasowanie i zgłosi wyjątek — i dokładnie tego chcemy, ponieważ gdybyśmy tworzyli jawne testy typów, zrobilibyśmy to w taki sam sposób. Nie zapisując testów typu samodzielnie i pozwalając Pythonowi na wykrycie niedopasowania, redukujemy ilość kodu, jaki musimy zapisać, jednocześnie zwiększając jego elastyczność.

Zmienne lokalne

Najciekawszą częścią tego przykładu są chyba zmienne. Okazuje się, że zmienna `res` wewnętrz funkcji `intersect` jest czymś, co w Pythonie nazywane jest *zmienną lokalną* — nazwą widoczną jedynie dla kodu wewnętrz definicji funkcji i istniejącą jedynie w czasie wykonywania tej funkcji. Tak naprawdę, ponieważ wszystkie nazwy *przypisywane* w dowolny sposób wewnętrz funkcji są domyślnie klasyfikowane jako zmienne lokalne, prawie wszystkie nazwy z funkcji `intersect` są zmiennymi tego typu.

- Zmienna `res` jest przypisywana w jawnym sposobie, dlatego jest zmienną lokalną.
- Argumenty przekazywane są przez przypisanie, dlatego również `seq1` i `seq2` są zmiennymi lokalnymi.
- Pętla `for` przypisuje elementy do zmiennej, zatem nazwa `x` również jest lokalna.

Wszystkie wymienione zmienne lokalne pojawiają się, gdy funkcja jest wywoływana, i znikają, gdy funkcja kończy działanie — instrukcja `return` na końcu funkcji `intersect` odsyła *obiekt* wynikowy, ale *nazwa res* znika. Z tego powodu zmienne funkcji nie zapamiętują wartości między wywołaniami; chociaż obiekt zwracany przez funkcję istnieje, zachowanie innych rodzajów informacji o jego stanie wymaga zastosowania innych technik. Aby jednak w pełni poznać pojęcia zmiennych lokalnych i stanu obiektów, musimy przejść do zagadnień związanych z zasięgami zmiennych, opisanych w rozdziale 17.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadzono podstawowe koncepcje związane z definicją funkcji — składnię i działanie instrukcji `def` oraz `return`, a także zachowanie wyrażeń wywołujących funkcję — oraz koncepcję i zalety polimorfizmu w funkcjach Pythona. Jak widzieliśmy, instrukcja `def` jest kodem wykonywalnym tworzącym obiekt funkcji w czasie wykonania. Kiedy później wywołamy tę funkcję, obiekty przekazywane są do niej za pomocą przypisania (warto przypomnieć, że w Pythonie przypisanie oznacza referencję do obiektu, co wewnętrznie, jak wiadomo z rozdziału 6., oznacza wskaźnik do miejsca w pamięci); obliczone wartości odsyłane są z powrotem przez instrukcję `return`. Zaczęliśmy również zapoznawać się z koncepcją zmiennych lokalnych oraz zasięgami zmiennych, których szczegóły zostaną omówione w rozdziale 17. Najpierw jednak pora na quiz.

Sprawdź swoją wiedzę — quiz

1. Jaki jest cel tworzenia funkcji?
2. W którym momencie Python tworzy funkcję?

3. Co zwraca funkcja, jeśli nie ma w niej instrukcji `return`?
4. Kiedy wykonywany jest kod zagnieżdzony wewnątrz instrukcji definicji funkcji?
5. Co jest złego w sprawdzaniu typów obiektów przekazywanych do funkcji?

Sprawdź swoją wiedzę – odpowiedzi

1. Funkcje są podstawowym sposobem unikania *powtarzalności* kodu w Pythonie. Faktoryzacja kodu w funkcje oznacza, że mamy tylko jedną kopię kodu operacji do uaktualnienia w przyszłości. Funkcje są również podstawową metodą *ponownego użycia kodu* w Pythonie — umieszczenie kodu w funkcjach powoduje, że można z niego korzystać później za pomocą wywołania w różnych programach. Funkcje pozwalają nam również na podzielenie złożonego systemu na części, którymi łatwiej jest zarządzać, a z których każda może być rozwijana niezależnie.
2. Funkcja jest tworzona, kiedy Python trafi na instrukcję `def` i ją wykona. Instrukcja ta tworzy obiekt funkcji i przypisuje go do nazwy funkcji. Normalnie dzieje się to, kiedy moduł ją zawierający importowany jest przez inny moduł (warto przypomnieć, że instrukcja `import` wykonuje kod pliku od góry do dołu, włączając w to wszystkie instrukcje `def`). Dzieje się tak jednak również wtedy, gdy instrukcja `def` wpisana zostanie w sesji interaktywnej lub zostaje zagnieżdzona w innych instrukcjach, takich jak `if`.
3. Funkcja domyślnie zwraca obiekt `None`, kiedy sterowanie wyjdzie poza jej ciało bez trafiaenia na instrukcję `return`. Takie funkcje zazwyczaj wywoływane są za pomocą instrukcji wyrażeń, gdyż przypisanie ich wyników (`None`) do zmiennych zwykle jest bezcelowe. Instrukcja `return` bez wyrażenia zwracającego rezultat działania funkcji również domyślnie zwraca obiekt `None`.
4. Ciało funkcji (kod zagnieżdzony wewnątrz instrukcji definicji funkcji) wykonywane jest, kiedy funkcja jest później wywoływana za pomocą wyrażenia wywołania. Ciało funkcji z każdym wywołaniem funkcji wykonywane jest na nowo.
5. Sprawdzanie typów obiektów przekazywanych do funkcji w praktyce niszczy jej elastyczność, ograniczając funkcję do działania jedynie na określonych typach. Bez takiego sprawdzania funkcja będzie najprawdopodobniej mogła przetworzyć całą gamę typów obiektów — każdy obiekt obsługujący interfejs oczekiwany przez funkcję będzie działał. (Pojęcie *interfejs* oznacza tutaj zbiór metod oraz operatorów wyrażeń wykonywanych przez kod funkcji).

[1] Takie polimorficzne zachowanie programu jest czasami nazywane *kaczym typowaniem*, które opiera się na założeniu, że kod programu nie powinien dbać o to, czy obiekt jest kaczką, tylko o to, czy kwacze. W takim scenariuszu program akceptuje każdy obiekt, który kwacze, niezależnie od tego, czy naprawdę jest kaczką, czy nie, przy czym sama implementacja mechanizmu kwakania należy do takiego obiektu. Jest to zasada, która stanie się jeszcze bardziej widoczna podczas omawiania zagadnień związanych z klasami w części VI tej książki. Oczywiście nasza kaczka to tylko bardzo obrazowa metafora (tak naprawdę jest to po prostu nowe opakowanie dla dosyć starej koncepcji programowania), ponieważ liczba zastosowań prawdziwie „kwaczącego” oprogramowania wydaje się być w praktyce bardzo mocno ograniczona (co powiedziawszy, zaczynam się nastawiać na przeglądanie setek zjadliwych e-maili od zaciętrewionych ornitologów...).

[2] Kod ten będzie działał zawsze, jeżeli będziemy sprawdzali część wspólną zawartości plików uzyskanych za pomocą funkcji `file.readlines()`. Może on jednak nie działać w przypadku części wspólnej plików wejściowych otwieranych w sposób bezpośredni, w zależności od implementacji operatora `in` w obiekcie pliku lub ogólnej iteracji. Po wczytaniu do końca pliki muszą generalnie być przewijane (na przykład za pomocą `file.seek(0)` lub kolejnego wywołania metody `open`), podobnie jak to ma miejsce w przypadku iteratorów jednoprzebiegowych. Jak zobaczymy w rozdziale 30. przy okazji omawiania przeciążania operatorów, obiekty implementują operator `in` albo udostępniając określoną metodę `__contains__`, albo obsługując ogólny protokół iteracji za pomocą metody `__iter__` lub starszej `__getitem__`. Klasy mogą dowolnie implementować takie metody w celu zdefiniowania, co będzie oznaczała iteracja dla ich danych.

Rozdział 17. Zasięgi

W rozdziale 16. wprowadzono podstawowe definicje i wywołania funkcji. Jak widzieliśmy, podstawowy model funkcji Pythona jest prosty w użyciu, jednak nawet proste przykłady funkcji sprawiły, że zaczęliśmy sobie zadawać pytania dotyczące znaczenia zmiennych w kodzie. W niniejszym rozdziale przedstawimy szczegóły dotyczące *zasięgów nazw* Pythona — miejsc, w których zmienne są definiowane i wyszukiwane. Podobnie jak pliki modułów, zasięgi pomagają zapobiegać kolizjom nazw w kodzie programu: nazwy zdefiniowane w jednej jednostce programowej nie kolidują z nazwami w innej.

Jak się przekonamy, miejsce, w którym zmienna jest przypisana w kodzie, ma kluczowe znaczenie w ustaleniu, co zmienna ta oznacza. Dowiemy się także, że korzystanie z zasięgów może mieć duży wpływ na wynik związanego z utrzymywaniem kodu. Na przykład nadużywanie zmiennych globalnych jest złym zwyczajem. Przekonasz się również, że zasięgi zmiennych mogą stanowić sposób na zachowanie *informacji o stanie* pomiędzy wywołaniami funkcji i w niektórych zastosowaniach stanowić ciekawą alternatywę dla klas.

Podstawy zasięgów w Pythonie

Skoro już zaczęliśmy tworzyć własne funkcje, musimy sprawdzić, co od strony formalnej oznaczają nazwy (zmienne) w Pythonie. Kiedy w programie użyjemy jakieś nazwy, Python tworzy, zmienia lub wyszukuje nazwę w czymś, co znane jest jako *przestrzeń nazw* (ang. *namespace*) — miejscu, w którym „życzą” nazwy. Kiedy mówimy o szukaniu wartości nazwy w odniesieniu do kodu, do przestrzeni nazw odnosi się *zasięg* (ang. *scope*). Lokalizacja przypisania nazwy w kodzie określa zasięg, w jakim nazwa ta jest w kodzie widoczna.

Prawie wszystko, co wiąże się z nazwami, w tym kwalifikacja do zasięgów, odbywa się w Pythonie w czasie przypisania. Jak widzieliśmy, nazwy w Pythonie zaczynają istnieć, kiedy pierwszy raz przypisuje się do nich wartości, co musi się odbyć przed ich pierwszym użyciem. Ponieważ nazw nie deklaruje się z wyprzedzeniem, Python wykorzystuje lokalizację przypisania do nazwy do *powiązania* jej z określona przestrzenią nazw. Innymi słowy, miejsce, w którym przypisujemy nazwę w kodzie źródłowym, określa przestrzeń nazw, w której nazwa ta będzie istniała, a tym samym również zasięg jej widoczności.

Poza łączeniem kodu w pakiety funkcje dodają do naszych programów dodatkową warstwę przestrzeni nazw, aby zminimalizować ryzyko wystąpienia kolizji pomiędzy zmiennymi o tych samych nazwach — *domyślnie wszystkie nazwy przypisane wewnętrz funkcji związane są tylko z przestrzenią nazw tej funkcji i żadną inną*. Oznacza to, że:

- Nazwy przypisane wewnętrz instrukcji `def` mogą być widoczne jedynie dla kodu wewnętrz tej instrukcji. Nie można się do nich odnosić w żaden sposób poza tą funkcją.
- Nazwy przypisane wewnętrz instrukcji `def` nie wchodzą w konflikt ze zmiennymi spoza tej instrukcji, nawet jeżeli tak samo się nazywają. Zmienna `X` przypisana poza definicją funkcji (na przykład w innej definicji funkcji lub na najwyższym poziomie pliku modułu) jest czymś zupełnie innym od zmiennej `X` przypisanej w tej funkcji.

W każdym przypadku zasięg zmiennej (czyli miejsce, w którym może ona zostać użyta) jest zawsze ustalany przez to, w którym miejscu kodu źródłowego zostaje ona przypisana, i nie ma nic wspólnego z tym, która funkcja wywołuje którą zmienną. Jak zresztą dowiemy się z

niniejszego rozdziału, zmienne można przypisywać w trzech różnych miejscach odpowiadających trzem różnym zasięgom:

- Jeżeli zmienna zostanie przypisana wewnątrz instrukcji `def`, staje się zmienną *lokalną* dla tej funkcji.
- Jeżeli zmienna przypisana jest wewnątrz instrukcji `def` zawierającej inną funkcję, staje się zmienną *nielokalną* dla tej funkcji.
- Jeżeli zmienna przypisana jest poza wszystkimi instrukcjami `def`, staje się zmienną *globalną* dla całego pliku.

Nazywamy to *zasięgiem leksykalnym*, ponieważ zasięg zmiennej uzależniony jest w całości od lokalizacji zmiennych w kodzie źródłowym plików programu, a nie od wywołań funkcji.

Na przykład w poniższym pliku modułu przypisanie `X = 99` tworzy zmienną *globalną* o nazwie `X` (widoczną w całym pliku), ale kolejne przypisanie, `X = 88`, tworzy zmienną *lokalną* `X` (widoczną jedynie wewnątrz instrukcji `def`).

```
X = 99                                     # Zmienna X globalna dla całego modułu

def func():
    X = 88                                    # Ta zmienna X jest lokalna dla funkcji; to
    inna zmienna
```

Pomimo iż obie zmienne noszą nazwę `X`, ich zasięgi sprawiają, że są one czymś innym. W rezultacie zasięg funkcji pozwala unikać konfliktów nazw w Pythonie i pomaga zamienić funkcje w bardziej samowystarczalne komponenty — nie musimy się martwić, czy nazwy zmiennych używane w ich kodzie występują gdzieś w innych miejscach programu.

Reguły dotyczące zasięgów

Zanim zaczeliśmy tworzyć funkcje, cały kod przez nas pisany był kodem najwyższego poziomu modułu (niezagnieżdżonym wewnątrz instrukcji `def`), dlatego wykorzystywane przez nas nazwy albo były częścią modułu, albo były wbudowane w samego Pythona (jak na przykład funkcja `open`). Technicznie rzecz biorąc, sesja interaktywna jest modułem o nazwie `_main_`, który wyświetla wyniki i nie zapisuje swojego kodu; pod każdym innym względem jest to jednak najwyższy poziom pliku modułu.

Funkcje udostępniają zagnieżdżone przestrzenie nazw (zasięgi), zawierające nazwy dla siebie lokalne, tak by zmienne te nie pozostawały w konflikcie z tymi spoza zasięgu funkcji (znajdującymi się w module czy innej funkcji). Funkcje definiują *zasięg lokalny*, natomiast moduły — *zasięg globalny*. Te dwa zasięgi wiążą się ze sobą w następujący sposób:

- **Moduł zawierający funkcję jest zasięgiem globalnym.** Każdy moduł jest zasięgiem globalnym, czyli przestrzenią nazw, w której znajdują się zmienne tworzone (przypisane) na najwyższym poziomie pliku modułu. Zmienne globalne stają się atrybutami obiektu modułu dla świata zewnętrznego, jednak wewnątrz samego modułu mogą być używane jako proste zmienne.
- **Zasięg globalny rozciąga się jedynie na jeden plik.** Nie daj się wprowadzić w błąd słowem „globalny” — zmienne przypisane na najwyższym poziomie pliku są globalne jedynie w odniesieniu do kodu tego jednego pliku. W Pythonie nie istnieje coś takiego jak jeden obejmujący wszystko globalny zasięg oparty na plikach. Zamiast tego zmienne dzielone są na moduły i jeżeli chcemy z nich skorzystać, musimy moduł zawsze w jawnym sposób zimportować. Kiedy słyszysz „globalny” w odniesieniu do Pythona, Twoim pierwszym skojarzeniem powinien być „moduł”.
- **Przypisane nazwy są lokalne, o ile nie zostaną zadeklarowane jako globalne lub nielokalne.** Domyslnie wszystkie nazwy przypisane wewnątrz definicji funkcji umieszczane są w zasięgu lokalnym (przestrzeni nazw powiązanej z wywołaniem funkcji).

Jeżeli chcesz przypisać zmienną istniejącą na najwyższym poziomie modułu zawierającego funkcję, możesz to zrobić, deklarując to w instrukcji global wewnątrz funkcji. Jeżeli potrzebujemy przypisać zmienną istniejącą w instrukcji def zawierającej inną funkcję, od Pythona 3.x możemy to zrobić, deklarując ją w instrukcji nonlocal.

- **Wszystkie pozostałe nazwy są lokalne dla zawierającej je funkcji, globalne lub wbudowane.** Nazwy, które nie zostały przypisane wewnątrz definicji funkcji, są lokalne dla zawierającego je zasięgu, zdefiniowane w otaczającym je poleceniu def, globalne (należące do przestrzeni nazw modułu) lub wbudowane (znajdujące się w udostępnianym przez Pythona module builtins).
- **Każde wywołanie funkcji tworzy nowy zasięg lokalny.** Za każdym razem, gdy wywołujesz funkcję, tworzysz nowy zasięg lokalny — to znaczy przestrzeń nazw, w której zwykle będą tworzone nazwy utworzone w tej funkcji. Każdą instrukcję def (i wyrażenie lambda) można traktować jako definiujące nowy zasięg lokalny, ale zasięg lokalny faktycznie odpowiada wywołaniu funkcji. Ponieważ Python pozwala funkcjom na wywoływanie samych siebie w pętli — jest to zaawansowana technika znana jako *rekurencja*, o której krótko wspomnieliśmy w rozdziale 9., kiedy omawialiśmy porównania — każde aktywne wywołanie otrzymuje własną kopię lokalnych zmiennych funkcji. Rekurencja jest również przydatna w funkcjach przeznaczonych do przetwarzania danych, których struktury nie można przewidzieć z góry; przyjrzymy się temu dokładniej w rozdziale 19.

Warto tu podkreślić kilka subtelności. Przede wszystkim powinieneś pamiętać, że kod wpisywany w wierszu poleceń konsoli Pythona również „znajduje się” w module, stąd zmienne utworzone w sesji interaktywnej również są częścią modułu i podlegają normalnym regułom dotyczącym zasięgów — są one globalne dla całej sesji interaktywnej.Więcej informacji o modułach znajdziesz w następnej części tej książki.

Warto także zauważyc, że *dowolny typ przypisania* wewnątrz funkcji klasyfikuje zmienną jako lokalną. Dotyczy to między innymi instrukcji ze znakiem =, zmiennych modułów w instrukcji import, zmiennych funkcji w instrukcji def, a także przekazywania argumentów. Jeżeli przypiszemy zmienną w jakikolwiek sposób wewnątrz instrukcji def, domyślnie stanie się ona lokalna dla tej funkcji.

Pamiętaj również o tym, że *modyfikacje obiektów w miejscu* nie klasyfikują zmiennych jako lokalnych — jest tak jedynie w przypadku właściwego przypisywania zmiennej. Jeżeli na przykład nazwa L zostanie przypisana do listy na najwyższym poziomie modułu, instrukcja taka jak L.append(X) wewnątrz funkcji nie zaklasyfikuje L jako zmiennej lokalnej, natomiast L = X już tak. W tym pierwszym przypadku zmieniamy obiekt listy, do którego odwołuje się zmienna L, a nie samą zmienną L. L zostanie jak zwykle odnaleziona w zasięgu globalnym, a Python z radością ją zmodyfikuje bez wymagania deklaracji global lub nonlocal. Jak zwykle należy jasno rozróżnić nazwy (zmienne) i obiekty — modyfikacja obiektu nie jest przypisaniem go do zmiennej.

Rozwiązywanie nazw — reguła LEGB

Jeżeli omówione wcześniej kwestie nie są wystarczająco jasne, warto dodać, że sprowadzają się one do trzech prostych reguł. W instrukcji def:

- *Przypisania* nazw domyślnie tworzą lub modyfikują nazwy lokalne.
- *Referencje* do nazw przeszukują co najwyżej cztery zasięgi — lokalny, następnie funkcji zawierających tę funkcję (jeżeli takie istnieją), globalny i na końcu wbudowany.
- Nazwy zadeklarowane w wyrażeniach global i nonlocal odwzorowują przypisane nazwy odpowiednio na zasięg modułu zawierającego oraz funkcji zawierającej.

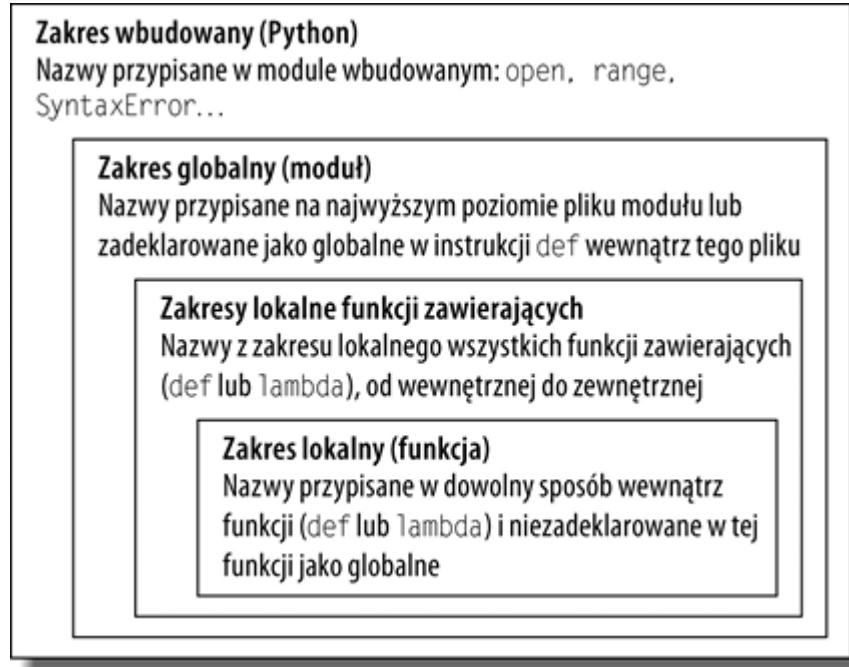
Innymi słowy, wszystkie nazwy przypisywane wewnątrz instrukcji def funkcji (lub wyrażenia lambda, z którym spotkamy się nieco później) są domyślnie lokalne. Funkcje mogą swobodnie wykorzystywać lokalne nazwy przypisane w funkcjach je zawierających, a także nazwy z

zasięgu globalnego, jednak by je zmodyfikować, muszą deklarować zmienne nielokalne oraz globalne.

Rozwiązywanie konfliktów w zasięgu nazw Pythona czasami nazywane jest *regułą LEGB* — od angielskich nazw kolejnych zasięgów:

- Kiedy wewnętrz funkcji użyjemy nazwy bez kwalifikatora, Python przeszuka cztery zasięgi — lokalny (*L*, od ang. *local*), następnie zasięg lokalny instrukcji `def` lub wyrażeń `lambda` zawierających daną funkcję (*E*, od ang. *enclosing*), później globalny (*G*, od ang. *global*), a na końcu wbudowany (*B*, od ang. *built-in*), zatrzymując się w pierwszym miejscu, w którym nazwa ta zostanie odnaleziona. Jeżeli nazwa nie zostanie znaleziona, Python zgłasza błąd.
- Kiedy przypisujemy zmienną wewnętrz funkcji (zamiast odnieść się do niej w wyrażeniu), Python zawsze tworzy lub modyfikuje tę zmienną w zasięgu lokalnym, o ile nie została ona w tej funkcji zadeklarowana jako globalna lub nielokalna.
- Kiedy przypisujemy zmienną poza jakąkolwiek funkcją (na przykład na najwyższym poziomie pliku modułu czy w sesji interaktywnej), zasięg lokalny jest tym samym co zasięg globalny — przestrzenią nazw modułu.

Ponieważ nazwy muszą być przypisane, zanim będą mogły być użyte (jak pisaliśmy o tym w rozdziale 6.), w tym modelu nie ma komponentów automatycznych: przypisania zawsze jednoznacznie określają zasięgi poszczególnych nazw. Na rysunku 17.1 przedstawiono cztery zasięgi Pythona. Warto zauważyć, że druga warstwa przeszukiwania — zasięg *E* (czyli zasięg instrukcji `def` lub wyrażeń `lambda` funkcji zawierającej kolejną funkcję) — może z technicznego punktu widzenia odpowiadać większej liczbie warstw. Taka sytuacja pojawia się tylko wtedy, gdy zagnieźdzamy funkcje w innych funkcjach, a w wersji 3.x może być rozszerzona za pomocą instrukcji `nonlocal`^[1].



Rysunek 17.1. Reguła przeszukiwania zasięgów LEGB. Kiedy w kodzie pojawia się odwołanie do zmiennej, Python szuka tej zmiennej w następującej kolejności: zasięg lokalny, zasięg lokalny funkcji zawierających tę funkcję, zasięg globalny i na końcu zasięg wbudowany. Wygrywa pierwsze wystąpienie. Zasięg zazwyczaj uzależniony jest od miejsca przypisania zmiennej w

kodzie. W Pythonie 3.x deklaracje nonlocal mogą także wymusić odwzorowanie zmiennych na zasięgi funkcji zawierającej, bez względu na to, czy są one przypisane

Należy również pamiętać, że reguły te mają zastosowanie jedynie do prostych nazw *zmiennych* (takich jak `spam`). W piątej i szóstej części książki zobaczymy, że kwalifikowane nazwy *atrybutów* (takie jak `object.spam`) istnieją w poszczególnych obiektach i obowiązują je zupełnie inne reguły wyszukiwania niż omówione tutaj koncepcje związane z zasięgami. Referencje do atrybutów (nazw następujących po kropce) przeszukują *obiekt* lub *obiekty*, a nie zasięgi, i mogą wywołać coś, co w zorientowanym obiektowo modelu programowania w Pythonie nazywamy *dziedziczeniem* (o którym będziemy pisać w VI części książki).

Inne zasięgi Pythona — przegląd

Choć do tej pory jeszcze nie zdążyliśmy o nich wspomnieć, to jednak warto zauważyc, że w Pythonie istnieją trzy dodatkowe rodzaje zasięgów — zmienne tymczasowe w pętli w niektórych wyrażeniach składanych, zmienne odniesienia wyjątków w niektórych modułach obsługi `try` oraz lokalne zasięgi w instrukcjach klas. Pierwsze dwa z nich to przypadki szczególne, które rzadko wpływają na prawdziwy kod, a trzeci podlega ogólnej zasadzie wyszukiwania LEGB.

Większość bloków instrukcji i innych konstrukcji nie lokalizuje używanych przez nie nazw, z następującymi wyjątkami specyficznymi dla wersji:

- Zmienne składane — zmienna X używana w odniesieniu do bieżącego elementu iteracji w wyrażeniu składanym, takim jak `[X for X in I]`. Ponieważ mogą kolidować z innymi nazwami i odzwierciedlać stan wewnętrzny generatorów, w wersji 3.x takie zmienne są lokalne dla samego wyrażenia we wszystkich formach elementów składanych, takich jak generatory, listy, zbiory i słowniki. W wersji 2.x są one lokalne dla wyrażeń generatorów, zbiorów i słowników składanych, ale nie dla list składanych, które odwzorowują ich nazwy na zasięg poza wyrażeniem. Dla porównania: pętle `for` nigdy i w żadnej z wersji języka Python nie ograniczały zasięgu swoich zmiennych do bloku instrukcji. Więcej informacji i przykładów znajdziesz w rozdziale 20.
- Zmienne wyjątków — zmienna X używana do odwoływanego się do zgłoszonego wyjątku w klauzuli obsługi wyjątku instrukcji `try`, na przykład `except E as X`. Ponieważ mogą one odroczyć działanie mechanizmu odzyskiwania nieużywanych zasobów pamięci, w wersji 3.x takie zmienne są lokalne dla tego bloku `except` i w praktyce są usuwane po zakończeniu działania bloku (nawet jeżeli używałeś go wcześniej w kodzie!). W wersji 2.x takie zmienne pozostają aktywne po wykonaniu instrukcji `try`. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 34.

W zasadzie oba przedstawione konteksty rozszerzają regułę LEGB, zamiast ją modyfikować. Na przykład zmienne składane są po prostu powiązane z zagnieźdzonym i specjalnym zasięgiem; inne nazwy przywoływane w tych wyrażeniach są traktowane zgodnie ze zwykłymi regułami wyszukiwania LEGB.

Warto również zauważyc, że instrukcja `class`, którą spotkamy w części VI, tworzy również nowy *zasięg lokalny* dla nazw przypisanych wewnątrz najwyższego poziomu jej bloku. Jeżeli chodzi o `def`, nazwy przypisane w klasie nie kolidują z nazwami w innych miejscach i są zgodne z regułą wyszukiwania LEGB, gdzie blok klasy znajduje się na poziomie „L”. Podobnie jak moduły i importy, nazwy te przekształcają się w atrybuty obiektów klasy po zakończeniu działania instrukcji `class`.

Jednak w przeciwieństwie do funkcji nazwy z klasy nie są tworzone dla każdego wywołania: wywołania obiektów klasy generują *instancje klasy*, które dziedziczą nazwy przypisane w klasie i rejestrują stan jej obiektów jako atrybuty. Jak zobaczyłeś również w rozdziale 29., chociaż reguła LEGB służy do rozpoznawania nazw używanych zarówno na najwyższym poziomie samej klasy, jak i na najwyższym poziomie metod w niej zagnieźdzonych, same klasy są *pomijane* przez wyszukiwania zasięgów — ich nazwy należy pobrać jako atrybuty obiektu. Ponieważ Python w poszukiwaniu przywoływanych nazw przeszukuje funkcje otaczające, ale nie obejmuje klas, reguła LEGB nadal ma zastosowanie do kodu OOP.

Przykład zasięgu

Przyjrzyjmy się większemu przykładowi demonstrującemu koncepcje związane z zasięgami. Założymy, że w pliku modułu zapiszemy poniższy kod:

```
# Zasięg globalny

X = 99                                # X i func przypisane w module:
globalne

def func(Y):                            # Y i Z przypisane w funkcji:
    lokalne

    # Zasięg lokalny

    Z = X + Y                          # X jest globalne

    return Z

func(1)                                  # func w module: wynik = 100
```

Moduł i umieszczona w nim funkcja wykorzystują kilka zmiennych. Za pomocą reguł zasięgu Pythona możemy zmienne sklasyfikować w następujący sposób:

Nazwy globalne — X, func

Zmienna X jest globalna, ponieważ zostaje przypisana na najwyższym poziomie pliku modułu. Można się do niej odwoływać z wnętrza funkcji jak do prostej, niekwalifikowanej nazwy bez konieczności deklarowania jej jako zmiennej globalnej. Nazwa func jest globalna z tego samego powodu; instrukcja def przypisuje obiekt funkcji do nazwy func na najwyższym poziomie modułu.

Nazwy lokalne — Y, Z

Zmienne Y oraz Z są lokalne dla funkcji (i istnieją tylko w czasie jej działania), ponieważ do obu przy definiowaniu funkcji przypisaliśmy wartości — do Z za pomocą instrukcji =, natomiast do Y dlatego, że argumenty zawsze przekazywane są przez przypisanie.

Celem takiego schematu rozdzielania nazw jest to, że zmienne lokalne służą jako *tymczasowe* nazwy potrzebne tylko w czasie działania funkcji. W powyższym przykładzie argument Y oraz wynik dodawania Z istnieją jedynie wewnątrz funkcji. Nazwy te nie wchodzą w konflikt z przestrzenią nazw zawierającą funkcję modułu (ani z żadną inną funkcją). W rzeczywistości zmienne lokalne są usuwane z pamięci po zakończeniu działania funkcji, a zasoby obiektów, do których się odwołują, mogą zostać zwolnione, jeżeli nie zostaną przywołane gdzie indziej. Jest to automatyczny, wewnętrzny etap działania, ale pomaga zminimalizować wymagania dotyczące pamięci.

Rozróżnienie zmiennych na lokalne i globalne sprawia również, że funkcje łatwiej jest zrozumieć, ponieważ większość nazw wykorzystywanych przez funkcję pojawia się tylko w niej, a nie w dowolnym miejscu modułu. Ponieważ możemy być pewni, że nazwy lokalne nie zostaną zmodyfikowane przez jakąś inną funkcję programu, zazwyczaj pozwala to na łatwiejsze debugowanie i modyfikowanie kodu. Funkcje są samodzielnymi jednostkami oprogramowania.

Zasięg wbudowany

Dotychczas zasięg wbudowany traktowaliśmy jak abstrakcję, jednak w rzeczywistości cała sprawa jest łatwiejsza, niż można przypuszczać. Tak naprawdę zasięg wbudowany to po prostu wbudowany moduł o nazwie `builtins`; aby jednak użyć tego modułu, trzeba go zimportować, ponieważ sama nazwa `builtins` nie jest zmienną wbudowaną.

Naprawdę, nie żartuję! Zasięg wbudowany został zaimplementowany w wersji 3.x jako jeden moduł biblioteki standardowej o nazwie `builtins`, jednak nazwa ta nie została umieszczona w zasięgu wbudowanym, dlatego aby przejrzeć zawartość tego modułu, trzeba go zaimportować. Po zrobieniu tego możemy wywołać funkcję `dir` w celu sprawdzenia, jakie nazwy zostały zdefiniowane w tym module. W Pythonie 3.3 wygląda to tak (o sposobie użycia w wersji 2.x powiemy już za chwilę):

```
>>> import builtins  
>>> dir(builtins)  
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',  
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',  
...wiele kolejnych nazw zostało celowo pominiętych...  
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed',  
'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',  
'super', 'tuple', 'type', 'vars', 'zip']
```

Nazwy z tej listy składają się na zasięg wbudowany w Pythonie. Mniej więcej pierwsza połowa to wbudowane wyjątki, natomiast druga to wbudowane funkcje. Na liście tej można także znaleźć nazwy specjalne, takie jak `None`, `True` oraz `False`, choć w wersji 3.x są one traktowane jako słowa zarezerwowane. Ponieważ Python automatycznie przeszukuje ten moduł jako ostatni (zgodnie z regułą przeszukiwania zasięgów LEGB), wszystkie nazwy z tej listy otrzymujemy „za darmo” — co oznacza, że możemy z nich korzystać bez konieczności importowania żadnych modułów. Z tego powodu do funkcji wbudowanej można się tak naprawdę odwołać na dwa sposoby — korzystając z reguły LEGB lub ręcznie importując moduł `builtins`.

```
>>> zip # W normalny sposób  
<class 'zip'>  
>>> import builtins # W bardziej skomplikowany sposób  
>>> builtins.zip  
<class 'zip'>  
>>> zip is builtins.zip # Ten sam obiekt, inne wyszukiwanie  
True
```

To drugie rozwiązanie przydaje się czasami przy realizacji bardziej zaawansowanych zadań, o których wspominamy w ramkach w dalszej części tego rozdziału.

Przeddefiniowanie wbudowanych nazw: lepiej czy gorzej?

Uważny Czytelnik z pewnością dostrzeże również, że ponieważ procedura przeszukiwania zasięgów LEGB pobiera *pierwszą* odnalezioną nazwę, nazwy z zasięgu lokalnego mogą nadpisywać zmienne o tej samej nazwie w zasięgu globalnym i zasięgu wbudowanym, a nazwy globalne mogą nadpisywać nazwy wbudowane.

Funkcja może na przykład utworzyć zmienną lokalną o nazwie `open`, przypisując do niej jakąś wartość:

```
def hider():  
    open = 'mielonka' # Zmienna lokalna, ukrywa wbudowaną  
    zmienną  
    ...
```

```
open('data.txt') # Błąd: Takie polecenie nie otworzy  
pliku w tym zasięgu!
```

Spowoduje to jednak ukrycie wbudowanej funkcji `open`, która działa we wbudowanym (zewnętrznym) zakresie, tak że nazwa `open` nie będzie już działać z funkcją otwierającą pliki — jest to teraz ciąg znaków, a nie funkcja otwierająca pliki. Nie stanowi to problemu, jeżeli nie musisz tutaj otwierać plików, ale powoduje błąd przy próbie otwarcia pliku przy użyciu tej nazwy.

Może się to zdarzyć nawet w sesji interaktywnej, która działa jako globalny zasięg modułu:

```
>>> open = 99 # Przypisanie w zasięgu globalnym ukrywa  
tutaj wbudowaną funkcję
```

W zasadzie nie ma *nic złego* w używaniu wbudowanych nazw zmiennych do własnych celów, o ile nie potrzebujesz ich oryginalnej, wbudowanej wersji. W końcu, gdyby były one naprawdę niedostępne, musielibyśmy zapamiętać całą wbudowaną listę nazw i traktować wszystkie znajdujące się na niej nazwy jako zastrzeżone. Przy ponad 140 nazwach w tym module w wersji 3.3 byłoby to zbyt restrykcyjne i zniechęcające:

```
>>> len(dir(builtins)), len([x for x in dir(builtins) if not  
x.startswith('__')])  
(148, 142)
```

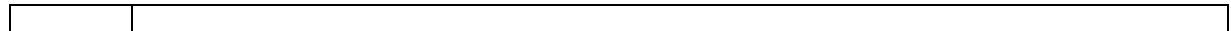
W praktyce podczas pracy nad zaawansowanymi projektami mogą zdarzyć się sytuacje, w których możesz chcieć zastąpić wbudowaną nazwę poprzez przedefiniowanie jej w kodzie — na przykład aby zdefiniować niestandardową funkcję `open`, która będzie weryfikowała choćby próby dostępu (więcej szczegółowych informacji na ten temat znajdziesz w ramce „Jak wywrócić świat do góry nogami w Pythonie 2.x” w dalszej części tego rozdziału).

Mimo to redefiniowanie wbudowanej nazwy jest często błędem, a do tego nieprzyjemnym, ponieważ Python o tym nie ostrzega. Narzędzia takie jak *PyChecker* (więcej informacji na ten temat znajdziesz w internecie) mogą ostrzegać Cię przed takimi błędami, ale Twoją najlepszą obroną jest tutaj wiedza — nie zmieniaj definicji wbudowanych nazw Pythona. Jeżeli przypadkowo zmienisz przypisanie wbudowanej nazwy w sesji interaktywnej, to możesz albo zrestartować sesję, albo uruchomić instrukcję `del nazwa`, aby usunąć redefinicję nazwy z bieżącego zasięgu, a tym samym przywrócić oryginał.

Zwróć uwagę, że funkcje mogą w podobny sposób ukrywać zmienne globalne za pomocą zmiennych lokalnych o takich samych nazwach, ale jest to znacznie bardziej użyteczne rozwiązanie i w praktyce jest to w dużej mierze zadaniem lokalnych zasięgów — Twoje funkcje są niezależnymi przestrzeniami nazw, dzięki czemu minimalizują możliwość występowania potencjalnych konfliktów nazw:

```
X = 88 # Zmienna globalna X  
  
def func():  
    X = 99 # Zmienna lokalna X: ukrywa globalną, ale  
    właśnie o to nam tutaj chodzi!  
  
    func()  
  
    print(X) # Wyświetla 88: bez zmian
```

W powyższym kodzie instrukcja przypisania wewnętrz funkcji tworzy zmienną lokalną `X`, całkowicie odrębną od zmiennej globalnej `X` z modułu znajdującej się na zewnątrz funkcji, ale jedną z konsekwencji takiego rozwiązania jest to, że *nie można zmienić* nazwy znajdującej się poza funkcją bez dodania deklaracji `global` (lub `nonlocal`) do `def`, jak opisano w następnej sekcji.





Uwaga na temat zmiany w wersji: Teraz dopiero sprawy zaczynają się komplikować. Wykorzystany tutaj moduł `builtins` z Pythonem 3.x w Pythonie 2.x nosi nazwę `__builtin__`. A żeby było jeszcze śmieszniej, nazwa `__builtins__` (z „`s`” na końcu) jest z góry ustawiana w większości zasięgów globalnych, w tym sesji interaktywnej, i odwołuje się do modułu znanego pod nazwą `builtins` w wersji 3.x lub `__builtin__` w 2.x, więc często możesz używać `__builtins__` bez importowania, ale nie możesz uruchomić importu dla samej nazwy — jest to zmienna predefiniowana, a nie nazwa modułu.

Tym samym po zainportowaniu `builtins` wyrażenie `__builtins__ is builtins` zwraca w Pythonie 3.x wartość `True`, natomiast w Pythonie 2.x wyrażenie `__builtins__ is __builtin__` zwróci wartość `True` po zainportowaniu modułu `__builtin__`. Rezultat jest taki, że zazwyczaj możemy sprawdzić wbudowany zasięg, wykonując funkcję `dir(__builtins__)` bez importowania — i to zarówno w wersji 3.x, jak i 2.x, ale zalecanym rozwiązaniem jest jednak wykorzystywanie modułu `builtins` dla wszystkich działań w Pythonie 3.x, a modułu `__builtin__` do takich zadań w Pythonie 2.x. Kto powiedział, że dokumentowanie takich zawiłości ma być proste?

Jak wywrócić świat do góry nogami w Pythonie 2.x

Oto kolejna rzecz, jaką można zrobić w Pythonie, choć raczej się nie powinno. Ponieważ nazwy `True` i `False` w wersji 2.x są po prostu zmiennymi z zasięgu wbudowanego, a nie słowami zarezerwowanymi, możemy je przypisywać za pomocą instrukcji takich jak `True = False`. Nie ma się co przejmować, takim działaniem wcale nie zniszczymy logicznej spójności świata! Wykonanie takiego polecenia po prostu zmieni znaczenie słowa `True` dla zasięgu, w którym się pojawią. Wszystkie pozostałe zasięgi odnajdą oryginalne nazwy w zasięgu wbudowanym.

Większym dowcipem jest wpisanie w Pythonie 2.x instrukcji `__builtin__.True = False`, która zmienia `True` na `False` dla całego procesu Pythona! Takie rozwiązanie działa, ponieważ w Pythonie jest tylko jeden wbudowany zasięg, współużytkowany przez wszystkich jego klientów. W Pythonie 3.x wykonanie takiego polecenia jest niemożliwe, ponieważ `True` i `False` traktowane są jako prawdziwe słowa zarezerwowane, podobnie jak `None`. W wersji 2.x wprawia to jednak środowisko IDLE w dziwny stan, który powoduje rozpoczęcie procesu kodu użytkownika od nowa (innymi słowy, nie powinieneś próbować tego w domu!).

Taka technika jest jednak użyteczna zarówno dla celów zilustrowania modelu przestrzeni nazw, jak i dla autorów narzędzi, którzy muszą zmieniać funkcje wbudowane, takie jak `open`, aby dostosować ich działanie do własnych potrzeb. Przez ponowne przypisanie nazwy funkcji z wbudowanego zasięgu zamieniając ją na jej odpowiednik dla każdego modułu w procesie. Jeżeli to zrobisz, prawdopodobnie będziesz także musiał zapamiętać jej oryginalną wersję, aby mieć możliwość jej wywołania z poziomu swojej spersonalizowanej wersji — w ramce „Warto pamiętać — dostosowywanie funkcji `open`” pokażemy jeden ze sposobów na osiągnięcie tego celu, ale najpierw musimy omówić kilka zagadnień związanych z zagnieźdzaniem zasięgów oraz opcjami zachowania stanu. Warto również przypomnieć, że narzędzia zewnętrzne, takie jak PyChecker czy PyLint, ostrzegają nas przed często popełnianymi błędami programistycznymi, w tym przypadkowym przypisaniem do wbudowanej nazwy Pythona (w takich narzędziach to zjawisko często jest określone mianem „przesłaniania” funkcji wbudowanej), zatem uruchamianie kilku swoich pierwszych programów napisanych w języku Python za pomocą narzędzi takich jak wymienione wyżej jest całkiem niezłyym pomysłem.

Instrukcja `global`

Instrukcja `global` oraz jej krewny `nonlocal` z wersji 3.x są jedynymi elementami w Pythonie przypominającymi instrukcje deklaracji. Nie są to jednak deklaracje typu czy rozmiaru — to *deklaracje przestrzeni nazw*. Instrukcja `global` informuje Pythona, że funkcja planuje zmienić jedną lub więcej nazw globalnych — to znaczy nazw, które znajdują się w zasięgu danego modułu (inaczej mówiąc, w przestrzeni nazw danego modułu).

Wspominaliśmy już wcześniej o instrukcji `global`; oto kilka słów przypomnienia:

- Nazwy globalne to zmienne przypisane na najwyższym poziomie pliku modułu.
- Nazwy globalne muszą być deklarowane jedynie wtedy, gdy przypisywane są wewnątrz funkcji.
- Do nazw globalnych można się odwoływać wewnątrz funkcji bez ich deklarowania.

Innymi słowy, instrukcja `global` pozwala *modyfikować* zmienne znajdujące się poza instrukcją `def` na najwyższym poziomie pliku modułu. Jak zobaczymy później, instrukcja `nonlocal` działa prawie identycznie, jednak ma zastosowanie do zmiennych w zasięgu lokalnym zawierającej ją instrukcji `def`, a nie do zmiennych z otaczającego ją modułu.

Instrukcja `global` składa się ze słowa kluczowego `global`, po którym następuje jedna lub większa liczba nazw rozdzielonych przecinkami. Po przypisaniu lub odniesieniu wewnątrz ciała funkcji wszystkie wymienione nazwy zostaną odwzorowane na zasięg modułu zawierającego funkcję, jak w przykładzie niżej.

```
X = 88                                     # Zmienna globalna X
def func():
    global X
    X = 99                                     # Zmienna globalna X: poza def
func()
print(X)                                      # Wyświetla 99
```

Do powyższego przykładu dodajemy teraz deklarację `global`, tak by zmienna `X` wewnątrz instrukcji `def` odnosiła się do zmiennej `X` spoza tej instrukcji — w takim scenariuszu oba „iksy” wskazują na jedną i tę samą zmienną, stąd zmiana wartości zmiennej `X` wewnątrz funkcji zmienia również wartość `X` na zewnątrz tej funkcji. Poniżej przedstawiamy nieco bardziej skomplikowany przykład działania instrukcji `global`:

```
y, z = 1, 2                                # Zmienne globalne w module
def all_global():
    global x
    global y                                     # Deklaracja zmiennych
    x = y + z                                  # Nie trzeba przypisywać y i z –
    działa reguła LEGB
```

Tutaj zmienne `x`, `y` oraz `z` są globalne wewnątrz funkcji `all_global`. Zmienne `y` oraz `z` są globalne, ponieważ są przypisane poza funkcją, natomiast zmienna `x` jest globalna, ponieważ została wymieniona w instrukcji `global`, tak by w jawnym sposób odwzorowywać bezpośrednio zasięg globalny. Bez instrukcji `global` zmienna `x` byłaby uznawana za lokalną ze względu na miejsce przypisania.

Warto zwrócić uwagę na to, że zmienne `y` oraz `z` nie są deklarowane jako globalne. Reguły wyszukiwania LEGB Pythona automatycznie odnajdują je w module. Zmienna `x` może nie istnieć w module zawierającym funkcję przed jej wykonaniem. W tym przypadku przypisanie wewnątrz funkcji tworzy zmienną `x` w module.

Projektowanie programów: minimalizowanie stosowania zmiennych globalnych

Zagadnienia związane z funkcjami w ogóle, a ze zmiennymi globalnymi w szczególności, często rodzą wiele pytań dotyczących projektowania samego programu. Jak powinny komunikować się nasze funkcje? Choć niektóre z wymienionych niżej zagadnień staną się bardziej oczywiste, gdy zaczniesz pisać własne większe funkcje, kilka praktycznych wskazówek podanych teraz może Ci później zaoszczędzić wielu niepotrzebnych problemów. Ogólnie rzecz biorąc, funkcje powinny opierać się na argumentach i zamiast modyfikowania zmiennych globalnych powinny zwracać wyniki swojego działania, ale zanim przejdziemy dalej, muszę wyjaśnić, dlaczego tak jest.

Zmienne przypisane w funkcjach są domyślnie lokalne, jeśli zatem chcemy zmodyfikować zmienne spoza funkcji, musimy wstawić do funkcji dodatkowy kod (na przykład instrukcję `global`). Takie rozwiązanie jest celowe — tak, jak często ma to miejsce w Pythonie — jeżeli chcemy zrobić coś potencjalnie niewłaściwego, musimy się bardziej postarać. Choć czasami zmienne globalne są użyteczne, zmienne przypisywane w instrukcji `def` są domyślnie lokalne, ponieważ zazwyczaj jest to najlepsze rozwiązanie. Modyfikacja zmiennych globalnych może prowadzić do wielu powszechnie znanych problemów programistycznych — ponieważ wartości zmiennych uzależnione są od kolejności wywoływania odrębnych od siebie funkcji, programy mogą być trudniejsze do debugowania i analizowania.

Rozważmy na przykład następujący plik modułu.

```
X = 99

def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

Wyobraźmy sobie teraz, że naszym zadaniem jest modyfikacja lub ponowne wykorzystanie tego pliku modułu. Jaką wartość będzie miała zmienna `X`? To pytanie nie ma tak naprawdę znaczenia, dopóki nie wzbogacimy go o jakiś punkt odniesienia w *czasie* — wartość `X` zmienia się w czasie, ponieważ uzależniona jest od tego, która funkcja została wywołana jako ostatnia (czyli od czegoś, czego nie da się powiedzieć na podstawie tego pliku).

Rezultat jest taki, że by zrozumieć kod, musimy prześledzić sterowanie w *całym programie*. Jeżeli będziemy chcieli ten kod zmodyfikować lub ponownie wykorzystać, musimy pamiętać o całym programie jednocześnie. W tym przypadku nie możemy tak naprawdę użyć żadnej z funkcji bez skorzystania przy okazji z tej drugiej. Są one uzależnione od zmiennej globalnej (czyli z nią *powiązane*). Jest to wada zmiennych globalnych — zazwyczaj sprawiają, że kod trudniej jest zrozumieć i używa w porównaniu z kodem składającym się z samodzielnych, odrębnych funkcji opartych na zmiennych lokalnych.

Z drugiej strony, poza wykorzystywaniem zagnieżdżanych zasięgów, programowania zorientowanego obiektywnego i klas, zmienne globalne są chyba najprostszym sposobem zachowywania w Pythonie współdzielonej *informacji o stanie* (informacji, które funkcja musi pamiętać w celu użycia przy następnym wywołaniu) — zmienne lokalne znikają po zakończeniu działania funkcji, ale zmienne globalne już nie. Jak zobaczymy później, taki efekt możemy również osiągnąć za pomocą innych technik, które pozwalają na zachowywanie wielu kopii informacji o stanie, ale są one na ogół bardziej złożone niż przekazywanie wartości do zmiennych globalnych, które znakomicie sprawdza się w wielu prostych zastosowaniach.

Czasami w programach tworzone są wręcz osobne moduły przeznaczone do gromadzenia zmiennych globalnych; dopóki takie zachowanie jest oczekiwane, nie jest aż tak szkodliwe. Dodatkowo programy wykorzystujące wielowątkowość do równoległego przetwarzania danych uzależnione są od zmiennych globalnych — stają się one pamięcią współdzieloną między funkcjami wykonującymi równolegle wątki, dzięki czemu działają jak narzędzia do komunikacji^[2].

Na razie jednak, zwłaszcza jeżeli dopiero zaczynasz swoją przygodę z programowaniem, powinieneś oprzeć się pokusie wykorzystywania zmiennych globalnych, gdzie tylko się da — utrudniają one zrozumienie sposobu działania programu i ponowne wykorzystywanie fragmentów kodu oraz nie działają w przypadkach, gdy jedna kopia zapisanych danych nie wystarczy. Zamiast tego spróbuj komunikować się za pomocą przekazywanych argumentów i zwracanych wartości. Za sześć miesięcy zarówno Ty, jak i Twoi współpracownicy z pewnością będziecie zadowoleni, że dokonaliście takiego wyboru.

Projektowanie programów: minimalizowanie modyfikacji dokonywanych pomiędzy plikami

Poniżej kolejna kwestia związana z zasięgiem: choć *możemy* bezpośrednio modyfikować zmienne w innym pliku, zazwyczaj nie powinniśmy tego robić. Pliki modułów zostały przedstawione w rozdziale 3. i są omawiane bardziej szczegółowo w kolejnej części niniejszej książki. Aby zilustrować ich związek z zasięgami, rozważmy dwa poniższe pliki modułów.

```
# Plik first.py
X = 99
second.py
# Ten kod nie wie nic o pliku

# Plik second.py
import first
print(first.X)
pliku
# OK: referencja do zmiennej w innym
# Zmodyfikowanie jej może być
first.X = 88
subtelne i niejawne
```

Pierwszy moduł definiuje zmienną X, drugi wyświetla ją, a następnie modyfikuje przez przypisanie. Warto zwrócić uwagę na to, że by pobrać zmienną z pierwszego modułu, musimy najpierw zimportować go do drugiego. Jak wiemy, każdy moduł jest odrębną przestrzenią nazw (pakietem zmiennych) i by z jednego modułu zobaczyć zawartość drugiego, musimy najpierw go zimportować. To właśnie najważniejsza cecha modułów — dzięki segregowaniu zmiennych według plików pozwalają one uniknąć konfliktów w zasięgach nazw pomiędzy plikami, podobnie jak zmienne lokalne zapobiegają powstawaniu konfliktów nazw między funkcjami.

Tak naprawdę w kontekście tematyki niniejszego rozdziału można jednak powiedzieć, że zasięg globalny pliku modułu *staje się* przestrzenią nazw atrybutów obiektu modułu po zimportowaniu. Plik importujący automatycznie zyskuje dostęp do wszystkich zmiennych globalnych pliku importowanego, ponieważ jego zasięg globalny po zimportowaniu *staje się* przestrzenią nazw atrybutów obiektu.

Po zimportowaniu pierwszego modułu drugi moduł wyświetla jego zmienną, a następnie przypisuje do niej nową wartość. Odwołanie się do zmiennej modułu w celu wyświetlenia jej jest w porządku — w taki sposób moduły normalnie łączą się ze sobą w większy system. Problem przypisania do first.X polega jednak na tym, że jest ono zbyt niejawne — osoby utrzymujące pierwszy moduł czy wykorzystujące go ponownie nie mają pojęcia, że jakiś daleki od niego inny

moduł z łańcucha importowania może zmodyfikować ich zmienną X w czasie wykonywania. Tak naprawdę drugi moduł może się znajdować w innym katalogu i być trudny do zauważenia.

Choć takie modyfikowanie zmiennych z różnych plików jest w Pythonie zawsze możliwe, zazwyczaj są to o wiele bardziej subtelne zmiany, niż byśmy tego chcieli. Powoduje to powstanie zbyt mocnego związku między dwoma plikami — ponieważ oba uzależnione są od wartości zmiennej X, trudno jest zrozumieć, czy można użyć ponownie jednego pliku bez drugiego. Takie niejawne zależności między plikami mogą prowadzić w najlepszym przypadku do tworzenia sztywnego, nieelastycznego kodu, a w najgorszym — do powstawania błędów.

I tutaj najlepszą receptą na rozwiązywanie problemu jest powstrzymanie się od takiego działania. Najlepszą metodą komunikacji ponad granicami plików jest wywoływanie funkcji, przekazywanie argumentów i otrzymywanie z powrotem wartości. W tym konkretnym przypadku lepiej byłoby utworzyć *funkcję akcesora*, która byłaby odpowiedzialna za tę zmianę.

```
# Plik first.py
X = 99
def setX(new):
    wprowadzane zmiany są jawne
    global X
    z jednego miejsca
    X = new
# Plik second.py
import first
first.setX(88)
# Wywołanie funkcji zamiast
# bezpośredniej zmiany wartości
```

Taki scenariusz wymaga większej ilości kodu i może się wydawać niewielką zmianą, ale jest to rozwiązywanie znacznie lepsze pod względem czytelności i możliwości utrzymywania kodu. Kiedy osoba analizująca kod pierwszego modułu zobaczy funkcję setX, będzie wiedziała, że jest to *punkt styku interfejsu* i będzie oczekiwana modyfikacja wartości zmiennej X. Innymi słowy, rozwiązanie to eliminuje element niespodzianki, który w projektach informatycznych rzadko jest czymś dobrym. Choć nie możemy zapobiec występowaniu zmian pomiędzy plikami, rozsądek nakazuje, by były one minimalizowane, o ile nie jest to szeroko akceptowane w programie.



Kiedy spotkamy się z klasami w części VI, poznasz podobne techniki kodowania akcesorów atrybutów. W przeciwieństwie do modułów klasy mogą również automatycznie przechwytywać pobieranie atrybutów, wykorzystując przeciążanie operatorów, nawet jeżeli klienci nie używają akcesorów.

Inne metody dostępu do zmiennych globalnych

Co ciekawe, ponieważ zmienne z zasięgu globalnego stają się atrybutami załadowanego obiektu modułu, możemy emulować instrukcję `global`, importując moduł ją zawierający i przypisując wartości do jego atrybutów, jak w poniższym przykładowym pliku modułu. Kod zamieszczony w tym pliku importuje najpierw moduł zawierający za pomocą jego nazwy, a następnie indeksuje `sys.modules`, czyli tabelę załadowanych modułów (więcej informacji na ten temat znajdziesz w rozdziałach 22. i 25.).

```
# Plik thismod.py
```

```

var = 99                                # Zmienna globalna == atrybut
modułu

def local():
    var = 0                                # Modyfikacja zmiennej lokalnej
var

def glob1():
    global var                            # Deklaracja zmiennej globalnej
(normalnej)

    var += 1                                # Modyfikacja zmiennej globalnej
var

def glob2():
    var = 0                                # Modyfikacja zmiennej lokalnej
var

    import thismod                         # Zaimportowanie siebie
    thismod.var += 1                        # Modyfikacja zmiennej globalnej
var

def glob3():
    var = 0                                # Modyfikacja zmiennej lokalnej
var

    import sys                             # Zaimportowanie tabeli
systemowej

    glob = sys.modules['thismod']          # Pobranie obiektu modułu (można
też użyć __name__)

    glob.var += 1                            # Modyfikacja zmiennej globalnej
var

def test():

    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)

```

Po wykonaniu tego kodu do zmiennej globalnej zostanie dodane 3 (jedynie pierwsza funkcja nie ma wpływu na tę zmienną).

```

>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102

```

Takie rozwiązanie działa i służy jako ilustracja tego, że zmienne globalne są odpowiednikami atrybutów modułów. Wymaga ono jednak więcej pracy niż umieszczenie w kodzie instrukcji `global`.

Jak widzieliśmy, instrukcja `global` pozwala nam modyfikować zmienne w modułach poza funkcjami. Ma ona również bliskiego krewniaka, instrukcję `nonlocal`, którą można także wykorzystać do zmiany nazw w funkcjach zawierających, jednak by zrozumieć, jak ona działa, musimy najpierw ogólnie omówić funkcje zawierające.

Zasięgi a funkcje zagnieżdżone

Dotychczas pomijałem jedną z części reguł zasięgu Pythona (celowo, ponieważ w praktyce spotyka się ją stosunkowo rzadko). Czas jednak przyjrzeć się uważniej literze `E` z reguły LEGB. Warstwa `E` jest stosunkowo nowa (została dodana w Pythonie 2.2); przybiera ona postać lokalnych zasięgów wszystkich instrukcji `def` zawierających funkcję. Zasięgi tego typu czasami nazywane są również *zasięgami zagnieżdżonymi statycznie*. Tak naprawdę jest to tylko zagnieżdżenie leksykalne — zagnieżdżone zasięgi odpowiadają fizycznie i składniowo zagnieżdżonym strukturom kodu w kodzie źródłowym programu.

Szczegóły dotyczące zasięgów zagnieżdżonych

Po dodaniu zasięgów funkcji zagnieżdżonych reguły wyszukiwania zmiennych stały się nieco bardziej skomplikowane. Wewnątrz funkcji:

- **Referencja (`X`)** wyszukuje zmienną `X` najpierw w zasięgu lokalnym (funkcji), a później w zasięgach lokalnych wszystkich funkcji leksykalnie zawierających tę funkcję w kodzie źródłowym, od wewnętrznej do zewnętrznej. Następnie szuka w bieżącym zasięgu globalnym (pliku modułu) i wreszcie w zasięgu wbudowanym (`module builtins`). Deklaracje `global` sprawiają, że wyszukiwanie rozpoczyna się zamiast tego w zasięgu globalnym (pliku modułu).
- **Przypisanie (`X = wartość`)** domyślnie tworzy bądź modyfikuje zmienną `X` w bieżącym zasięgu lokalnym. Jeżeli zmienna ta jest wewnątrz funkcji deklarowana jako *globalna*, przypisanie to zamiast tego tworzy lub modyfikuje zmienną `X` w zasięgu modułu zawierającego funkcję. Jeżeli z kolei zmienna ta jest wewnątrz funkcji deklarowana jako *nonlocal* (tylko w wersji 3.x), przypisanie modyfikuje zmienną `X` w zasięgu lokalnym najbliższej funkcji zawierającej.

Warto zauważyć, że deklaracja `global` nadal odwzorowuje zmienne na moduł je zawierający. Kiedy obecne są funkcje zagnieżdżone, do zmiennych z funkcji zawierających inne można się odnosić, jednak by je modyfikować, niezbędna jest deklaracja `nonlocal` (tylko w wersji 3.x).

Przykłady zasięgów zagnieżdżonych

Aby wyjaśnić te kwestie w praktyce, zilustrujemy je za pomocą prawdziwego kodu. Oto jak wygląda zasięg funkcji zamkijającej (umieść przedstawiony niżej kod w pliku skryptu lub wpisz go w sesji interaktywnej, aby uruchomić go na żywo):

```
X = 99                                     # Zmienna z zasięgu globalnego –
nieużywana

def f1():
    X = 88                                    # Zmienna lokalna z zasięgu
    zawierającego

    def f2():
        pass
```

```

print(X)                                # Referencja w zagnieżdżonej
instrukcji def

f2()

f1()                                     # Wyświetla 88 – zmienną lokalną
funkcji zawierającej

```

Po pierwsze jest to poprawny kod w Pythonie. Instrukcja `def` jest instrukcją wykonywalną; może pojawić się w każdym miejscu, w którym pojawiają się inne instrukcje, w tym wewnątrz innej instrukcji `def`. W powyższym kodzie zagnieżdżona instrukcja `def` wykonywana jest w momencie wywołania funkcji `f1`. Generuje ona funkcję i przypisuje ją do nazwy `f2` — lokalnej zmiennej z zasięgu lokalnego funkcji `f1`. W pewnym sensie `f2` jest tymczasową funkcją istniejącą jedynie w czasie wykonywania zawierającej ją funkcji `f1` i widoczną tylko dla kodu wewnątrz tej funkcji.

Warto jednak zwrócić uwagę na to, co dzieje się wewnątrz funkcji `f2`. Kiedy wyświetla ona zmienną `X`, odnosi się do `X` istniejącej w zasięgu lokalnym zawierającej ją funkcji `f1`. Ponieważ funkcje mają dostęp do nazw w fizycznie je zawierających instrukcjach `def`, zmienna `X` z funkcji `f2` jest automatycznie odwzorowana na zmienną `X` z funkcji `f1` za pomocą reguły wyszukiwania LEGB.

Wyszukiwanie w zasięgu zawierającym ma miejsce nawet wtedy, gdy funkcja zawierająca zakończyła już swoje działanie i zwróciła wartość. Na przykład kod przedstawiony poniżej definiuje funkcję tworzącą i *zwracającą* inną funkcję w dosyć powszechnie wykorzystywany sposób:

```

def f1():
    X = 88

def f2():
    print(X)                                # Pamięta X w zasięgu funkcji
    zawierającej

    return f2                               # Zwraca funkcję f2, ale jej nie
    wywołuje

action = f1()                             # Utworzenie i zwrócenie funkcji

action()                                  # Teraz wywołanie jej – wyświetla
88

```

W powyższym kodzie wywołanie `action` tak naprawdę uruchamia funkcję, którą podczas uruchamiania `f1` nazwaliśmy `f2`. Takie rozwiązanie działa, ponieważ funkcje są w Pythonie obiektami, podobnie jak wszystko inne, i mogą być przekazywane jako wartości zwracane z innych funkcji. Co najważniejsze, funkcja `f2` pamięta wartość zmiennej `X` z otaczającej funkcji `f1`, nawet jeżeli funkcja `f1` nie jest już aktywna — co prowadzi nas do następnego zagadnienia.

Funkcje fabrykujące: domknięcia

W zależności od tego, kogo zapytasz, takie zachowanie jest czasem nazywane *domknięciem* lub *funkcją fabrykującą* — to pierwsze opisuje technikę *programowania funkcyjnego*, a drugie oznacza *wzorzec projektowy*. Niezależnie od tego, jak go nazwiemy, obiekt funkcji, o którym mowa, pamięta wartości w otaczających zasięgach; obojętnie, czy te zasięgi są nadal obecne w pamięci, czy nie. W efekcie dołączane są obszary pamięci (np. przechowujące *informacje o stanie*), które są lokalne dla każdej kopii utworzonej funkcji zagnieżdżonej i często stanowią prostą alternatywę dla klas w tej roli.

Proste funkcje fabrykujące

Funkcje fabrykujące (nazywane też domknięciami) są czasami wykorzystywane przez programy, które muszą generować procedury obsługi zdarzeń „w locie” w odpowiedzi na warunki zmieniające się w czasie wykonywania. Na przykład wyobraź sobie graficzny interfejs użytkownika, który musi działać zgodnie z danymi wejściowymi użytkownika, niemożliwymi do przewidzenia w czasie tworzenia interfejsu GUI. W takich przypadkach potrzebujemy funkcji, która tworzy i zwraca inną funkcję, z informacjami, które mogą się różnić w zależności od wykonanej funkcji.

Aby to zilustrować w prosty sposób, rozważmy następującą funkcję, wpisaną w sesji interaktywnej (i pokazanej tutaj bez znaków wiersza kontynuacji „...”):

```
>>> def maker(N):
    def action(X):                      # Utworzenie i zwrócenie funkcji
        action
        return X ** N                  # Funkcja action zachowuje N z
    zasięgu funkcji zawierającej
    return action
```

Przedstawiony kod tworzy funkcję zewnętrzną, która po prostu generuje i zwraca funkcję zagnieżdzoną bez wywoływanego jej — funkcja `maker` tworzy funkcję `action` i zwraca ją bez jej wywoływanego jako rezultat swojego działania. Jeżeli wywołamy funkcję zewnętrzną:

```
>>> f = maker(2)                      # Przekazanie 2 do argumentu N
>>> f
<function maker.<locals>.action at 0x0000000002A4A158>
```

z powrotem otrzymamy referencję do wygenerowanej funkcji zagnieżdzonej, utworzonej przez wykonanie zagnieżdzonej instrukcji `def`. Jeżeli teraz wywołamy to, co otrzymaliśmy z funkcji zewnętrznej:

```
>>> f(3)                                # Przekazanie 3 do X, N pamięta 2:
3 ** 2
9
>>> f(4)                                # 4 ** 2
16
```

wywołana zostanie funkcja zagnieżdzona — nazywana w funkcji `maker` funkcją `action`. Innymi słowy, wywołujemy zagnieżdzoną funkcję, która została utworzona i zwrócona przez funkcję `maker`.

Najbardziej niezwykłym elementem tego przykładu jest jednak to, że funkcja zagnieżdzona *pamięta* liczbę całkowitą 2 — czyli wartość zmiennej `N` w funkcji `maker` — pomimo tego, że funkcja `maker` zwróciła wartość i zakończyła działanie, zanim jeszcze wywołaliśmy funkcję `action`. W rezultacie zmienna `N` z lokalnego zasięgu funkcji zawierającej jest przechowywana jako informacja o stanie i dodawana do wygenerowanej funkcji `action`, dzięki czemu otrzymujemy wynik będący kwadratem przekazanego argumentu.

Gdybyśmy teraz znowu wywołali funkcję zewnętrzną, otrzymalibyśmy z powrotem *nową* funkcję zagnieżdzoną z dołączonymi *innymi* informacjami o stanie. Argument zostałby tym samym podniesiony do sześciadanu, a nie do kwadratu, choć oryginalna funkcja nadal podnosi wartości do kwadratu.

```
>>> g = maker(3)                      # g pamięta 3, f pamięta 2
```

```

>>> g(4)                                # 4 ** 3
64
>>> f(4)                                # 4 ** 2
16

```

Powyższy kod działa, ponieważ każde wywołanie funkcji fabrykującej otrzymuje własny zbiór informacji o stanie. W naszym przypadku funkcja przypisana do zmiennej `g` pamięta liczbę 3, natomiast `f` pamięta liczbę 2, ponieważ każda z nich ma własne informacje o stanie zachowane przez zmienną `N` w funkcji `maker`.

Jest to jednak dosyć zaawansowana technika, którą w praktyce spotyka się raczej rzadko, z wyjątkiem kodu utworzonego przez programistów mających doświadczenie w funkcjonalnych językach programowania. Z drugiej strony zasięgi funkcji zawierającej często wykorzystywane są w wyrażeniach tworzących funkcje `lambda` (które zostaną omówione w dalszej części niniejszego rozdziału). Ponieważ są one wyrażeniami, prawie zawsze zagnieżdżane są wewnątrz struktury `def`. Na przykład wyrażenie `lambda` może zostać użyte zamiast `def` w następującym przykładzie:

```

>>> def maker(N):
    return lambda X: X ** N                      # funkcje lambda również
    zapamiętują informacje o stanie
>>> h = maker(3)
>>> h(4)                                         # ponownie 4 ** 3
64

```

Bardziej praktyczny przykład zastosowania funkcji fabrykującej znajdziesz w ramce „Warto pamiętać: dostosowywanie funkcji `open`” w dalszej części tego rozdziału. Przedstawione tam rozwiązanie używa podobnych technik do przechowywania informacji, które będą wykorzystane później w zasięgu funkcji zawierającej.



Uwaga do sposobu prezentacji przykładów: W tym rozdziale zacząłem zamieszczać przykłady interaktywne bez znaku zachęty wiersza kontynuacji „...”, które mogą pojawiać się w Twoim interfejsie (ale nie w środowisku IDLE). Od tej chwili będziemy korzystać z takiej formy zapisu, aby większe przykłady kodu były nieco łatwiejsze do wycinania i wklejania z e-booka lub innego nośnika elektronicznego. Zakładam, że rozumiesz już zasady tworzenia wcięć w Pythonie i masz pewne doświadczenie w pisaniu kodu Pythona — po prostu kody niektórych funkcji i klas prezentowanych w kolejnych rozdziałach mogą być zbyt duże, aby wpisywać je ręcznie.

Zamieszczamy również coraz więcej przykładów samodzielnego kodu lub plików i dowolnie przełączamy się między nimi a sesją interaktywną; gdy zobaczyesz znak zachęty `>>>`, oznacza to, że kod jest wpisywany interaktywnie i ogólnie można go wyciąć i wkleić do powłoki Pythona (pomijając znaki `>>>`). Jeżeli to się nie powiedzie, nadal możesz próbować uruchomić taki kod, wklejając go wiersz po wierszu lub edytując w pliku.

Funkcje fabrykujące kontra klasy, runda pierwsza

Niektórym użytkownikom *klasy*, opisane w całości w części VI tej książki, mogą wydawać się lepszym rozwiązaniem dla zachowywania stanów, ponieważ przechowują takie wartości w pamięci za pomocą bardziej jawnych przypisań atrybutów. Klasy bezpośrednio obsługują także dodatkowe narzędzia, których nie obsługują funkcje fabrykujące, takie jak dostosowywanie poprzez dziedziczenie i przeciążanie operatorów, a także bardziej naturalne wdrażanie wielu

zachowań w formie metod. Z powodu takich różnic klasy mogą lepiej sprawdzać się przy wdrażaniu bardziej kompletnych obiektów.

Mimo to funkcje fabrykujące często stanowią lżejszą i efektywniejszą alternatywę dla klas, gdy jedynym celem jest zapamiętywanie stanu. Zapewniają one zlokalizowane przechowywanie dla danych wymaganych przez pojedynczą, zagnieżdzoną funkcję. Jest to szczególnie prawdziwe, gdy w wersji 3.x użyjemy opisanej w kolejnych sekcjach instrukcji `nonlocal`, aby umożliwić zmiany stanu w zasięgu zawierającym (w wersji 2.x zasięgi zawierające są dostępne tylko do odczytu, a zatem mają bardziej ograniczone zastosowania).

Z szerszej perspektywy istnieje wiele sposobów, aby funkcje Pythona zachowywały stan między wywołaniami. Chociaż wartości normalnych zmiennych lokalnych zanikają po powrocie z funkcji, wartości można zachować od wywołania do wywołania w zmiennych globalnych, w atrybutach instancji klasy, w załączonych odniesieniach zasięgu (które poznaliśmy tutaj) oraz w wartościach domyślnych argumentów i atrybutach funkcji. Niektóre mogą również posiadać mutowalne argumenty domyślne (choć wielu użytkowników zapewne życzyłoby sobie, aby to nie było możliwe).



Funkcje fabrykujące można również utworzyć, gdy klasa jest zagnieżdzona w bloku instrukcji `def`: wartości lokalnych nazw funkcji fabrykującej są zachowywane przez odwołania w klasie lub w jednej z jej metod. Więcej szczegółowych informacji na temat klas zagnieżdzonych znajdziesz w rozdziale 29. Jak zobaczymy w późniejszych przykładach (np. dekoratorach z rozdziału 39.), zewnętrzne polecenie `def` w takim kodzie pełni podobną rolę: staje się fabryką klas i zapewnia zachowanie stanu dla zagnieżdzonej klasy.

W dalszej części tego rozdziału pokażemy rozwiązania alternatywne oparte na klasach i przedstawimy atrybuty funkcji, a pełne omówienie argumentów funkcji i ich wartości domyślnych znajdziesz w rozdziale 18. Aby ocenić, jak wartości domyślne sprawują się w zastosowaniach wymagających zachowania stanów, w następnej sekcji omówimy najważniejsze zagadnienia związane z tym tematem.

Zachowywanie stanu zasięgu zawierającego za pomocą argumentów domyślnych

We wcześniejszych wersjach Pythona (starszych niż 2.2) przedstawiony przed chwilą kod nie miałby racji bytu, ponieważ zagnieżdzone instrukcje `def` nie miały nic wspólnego z zasięgami — odwołanie do zmiennej wewnętrz funkcji `f2` spowodowałoby jedynie przeszukanie zasięgu lokalnego (tej funkcji), następnie globalnego (kod poza funkcją `f1`), a na koniec zasięgu wbudowanego. Ponieważ zasięgi funkcji zawierających `f2` zostały pominięte, otrzymalibyśmy błąd. Aby obejść ten problem, programiści zazwyczaj wykorzystywali *domyślne wartości argumentów* do przekazania (i zapamiętania) obiektów w zasięgu zawierającym.

```
def f1():
    x = 88

    def f2(x=x):                      # Pamięta x z zasięgu funkcji
        zawierającej z wartościami domyślnymi
            print(x)

    f2()
f1()                                # Wyświetla 88
```

Taki kod działa we wszystkich wydaniach Pythona i nadal można się z nim spotkać w wielu programach napisanych w tym języku. W skrócie, składnia `argument = wartość` w nagłówku

instrukcji `def` oznacza, że argument będzie miał podaną wartość domyślną, kiedy w wywołaniu nie zostanie do niego przekazana żadna prawdziwa wartość. Przedstawiona składnia służy tutaj do jawnego przypisania stanu zasięgu zawierającego w celu jego zachowania.

W zmodyfikowanej funkcji `f2` w powyższym kodzie, zapis `x=x` oznacza, że argument `x` będzie miał wartość domyślną równą `x` w zasięgu zawierającym — ponieważ drugie `x` zostanie obliczone, zanim Python wejdzie do zagnieżdzonej instrukcji `def`, nadal będzie się odnosiło do zmiennej `x` z funkcji `f1`. W rezultacie wartość domyślna pamięta, czym zmienna `x` była w funkcji `f1` (w tej sytuacji obiektem o wartości liczbowej 88).

Wszystko to jest stosunkowo skomplikowane i całkowicie uzależnione od momentu obliczenia wartości domyślnych. Tak naprawdę reguła przeszukiwania zasięgu zagnieżdzonego została w Pythonie dodana po to, by ukrócić wykorzystywanie wartości domyślnych w tej roli. Dzisiaj Python automatycznie pamięta wszystkie wartości wymagane w zasięgu zawierającym funkcję i może je wykorzystać w zagnieżdżonych instrukcjach `def`.

Oczywiście najlepszym rozwiązaniem w zdecydowanej większości przypadków jest po prostu unikanie zagnieżdzania instrukcji `def` wewnętrz instrukcji `def`, co pozwoli uprościć nasz program — zgodnie z regułami zen Pythona płaska struktura jest zawsze lepsza niż zagnieżdzona. Przykład pokazany poniżej jest odpowiednikiem wcześniejszego przykładu, w którym usunięto jedynie kwestie związane z zagnieżdzaniem. Warto zauważyć, że odwołanie robione z wyprzedzeniem jest poprawne — można wywoływać funkcję zdefiniowaną po funkcji zawierającej wywołanie, o ile druga instrukcja `def` wykonywana jest przed wywołaniem pierwszej funkcji. Kod wewnętrz instrukcji `def` nie jest wykonywany aż do momentu właściwego wywołania funkcji.

```
>>> def f1():
    x = 88
    f2(x)
    # Przekazanie x zamiast zagnieżdzania
    # Odwołanie z wyprzedzeniem jest OK

>>> def f2(x):
    print(x)
    # Płaska struktura jest lepsza niż
    # zagnieżdzona

>>> f1()
88
```

Jeżeli będziesz w taki sposób unikać zagnieżdzania, możesz właściwie zapomnieć o koncepcji zasięgów zagnieżdżonych w Pythonie. Z drugiej strony zagnieżdżone funkcje fabrykujące są dość powszechnie we współczesnym kodzie Pythona, podobnie jak funkcje `lambda`, które prawie naturalnie wydają się zagnieżdżone w blokach `def` i często opierają się na warstwie zagnieżdżonych zasięgów, jak wyjaśniono w następnej sekcji.

Zasięgi zagnieżdżone, wartości domyślne i wyrażenia lambda

Choć zasięgi funkcji zagnieżdżonych są dość często wykorzystywane w praktyce w samych instrukcjach `def`, zaczyna się pojawiać o wiele częściej, kiedy zaczynamy używać wyrażeń `lambda`. Z wyrażeniami `lambda` spotkaliśmy się już wcześniej tylko na chwilę, ale nie będziemy ich omawiać bardziej szczegółowo aż do rozdziału 19. Mówiąc jednak w skrócie, są to wyrażenia generujące nową funkcję, która ma być wywołana później, podobnie jak robi to instrukcja `def`. Ponieważ jest to jednak wyrażenie, można je umieszczać w miejscach, w których instrukcja `def` nie mogłaby się pojawić, na przykład w literałach list i słowników.

Podobnie do instrukcji `def`, wyrażenie `lambda` wprowadza nowy zasięg lokalny dla tworzonej przez siebie funkcji. Dzięki warstwie zasięgu zawierającego wyrażenia `lambda` widzą wszystkie zmienne istniejące w funkcjach, w których są utworzone. Z tego powodu poniższy kod, będący wariacją na temat przedstawionej wcześniej funkcji fabrykującej, działa poprawnie — tylko dzięki zastosowaniu nowych reguł dotyczących zasięgów zagnieżdżonych.

```
def func():
    x = 4
    action = (lambda n: x ** n)           # x pamiętałe z zasięgu
    zawierającego instrukcję def
    return action
x = func()
print(x(2))                            # Wyświetla 16, czyli 4 ** 2
```

Przed wprowadzeniem zasięgów funkcji zagnieżdżonych programiści wykorzystywali argumenty domyślne i przekazywali za ich pomocą wartości z zasięgów zawierających do wyrażeń `lambda` (podobnie jak dla instrukcji `def`). Na przykład poniższe rozwiązanie działa we wszystkich wersjach Pythona.

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n)      # Ręczne przekazanie x
    return action
```

Ponieważ `lambda` jest wyrażeniem, w naturalny sposób zagnieżdżana jest wewnątrz instrukcji `def`. Z tego powodu wyrażenie to chyba najbardziej zyskało na dodaniu zasięgu funkcji zawierającej do reguł wyszukiwania. W większości przypadków przekazywanie wartości do wyrażenia `lambda` za pomocą wartości domyślnych nie jest już konieczne.

Zmienne pętli mogą wymagać wartości domyślnych, a nie zasięgów

Istnieje jeden znaczący wyjątek od reguły przedstawionej powyżej (i jest to jednocześnie powód, dla którego pokazałem technikę wykorzystującą domyślne wartości argumentów): jeżeli wyrażenie `lambda` lub instrukcja `def` zdefiniowane są wewnątrz funkcji zagnieżdżonej w pętli, a funkcja zagnieżdżona zawiera odniesienie do zmiennej z zasięgu zawierającego, która modyfikowana jest przez tę pętlę, wszystkie funkcje wygenerowane w pętli będą miały tę samą wartość — taką, jaką zmienna miała w ostatniej iteracji pętli. W takich przypadkach nadal musisz używać wartości domyślnych, aby zapisać *bieżący* wartość zmiennej.

Może to wydawać się dość niejasne, ale w praktyce zdarza się częściej, niż myślisz, szczególnie w kodzie, który generuje funkcje obsługi wywołania zwrotnego dla wielu widżetów w graficznych interfejsach użytkownika — na przykład procedury obsługi kliknięć przycisków dla wszystkich przycisków w rzędzie. Jeżeli są tworzone w pętli, być może trzeba zachować ostrożność, aby zapisać stan z ustawieniami domyślnymi, w przeciwnym razie wszystkie wywołania zwrotne przycisków mogą zakończyć się tak samo.

Na przykład poniższy kod próbuje utworzyć listę funkcji pamiętających aktualną wartość zmiennej i z zasięgu zawierającego.

```
>>> def makeActions():
    acts = []
```

```

for i in range(5):                                # Próbuje zapamiętać każdą
wartość i...
    acts.append(lambda x: i ** x)      # ...ale wszystkie pamiętają tę
samą, ostatnią wartość i!
return acts

>>> acts = makeActions()
>>> acts[0]
<function makeActions.<locals>.<lambda> at 0x000000000002A4A400>

```

Takie rozwiązanie jednak nie działa — ponieważ zmienna z zasięgu zawierającego wyszukiwana jest przy okazji późniejszego *wywołania* funkcji zagnieżdżonych, wszystkie one w rezultacie zapamiętają tę samą wartość (wartość, jaką zmienna pętli miała przy *ostatniej* iteracji) — gdy przekazujemy argument potęgi 2 w każdym z poniższych wywołań, dla każdej funkcji z listy otrzymujemy w rezultacie liczbę 4 do kwadratu, ponieważ we wszystkich funkcjach zmienna i ma tę samą wartość.

```

>>> acts[0](2)                                # Wszystkie to 4 ** 2, wartość
ostatniego i = 4
16
>>> acts[1](2)                                # Powinno być 1 ** 2 (1)
16
>>> acts[2](2)                                # Powinno być 2 ** 2 (4)
16
>>> acts[4](2)                                # Tylko w tym przypadku wynik jest
prawidłowy 4 ** 2 (16)
16

```

To jedyny przypadek, gdy musimy przechować wartości z zasięgu funkcji zawierającej w sposób jawnym za pomocą argumentów domyślnych zamiast referencji do zasięgu funkcji zawierającej. By ten kod działał, musimy przekazać *aktualną* wartość zmiennej zasięgu funkcji zawierającej za pomocą domyślnej wartości argumentu. Ponieważ wartości domyślne obliczane są przy *tworzeniu* funkcji zagnieżdżonej (a nie przy jej późniejszym *wywołaniu*), każda pamięta swoją własną wartość dla zmiennej i.

```

>>> def makeActions():
    acts = []
    for i in range(5):                                # Użycie domyślnych wartości
argumentów
        acts.append(lambda x, i=i: i ** x) # Zapamiętanie bieżącej wartości
zmiennej i
    return acts

>>> acts = makeActions()
>>> acts[0](2)                                # 0 ** 2

```

```
0
>>> acts[1](2)                                # 1 ** 2
1
>>> acts[2](2)                                # 2 ** 2
4
>>> acts[4](2)                                # 4 ** 2
16
```

Wydaje się, że jest to artefakt implementacji, który jest podatny na zmiany i może stać się dla Ciebie bardziej istotny, gdy zaczniesz pisać większe programy. Więcej o ustawieniach domyślnych powiemy w rozdziale 18., a o wyrażeniach `lambda` w rozdziale 19., dlatego do niniejszego podrozdziału możesz również wrócić później[\[3\]](#).

Dowolne zagnieżdżanie zasięgów

Przed zakończeniem tej dyskusji należy zauważyc, że zakresy mogą być dowolnie zagnieżdżane, ale przy odwoływaniu się do nazw przeszukiwane są tylko instrukcje `def` funkcji zawierających (a nie klasy, opisane w części VI):

```
>>> def f1():
    x = 99
    def f2():
        def f3():
            print(x)                      # Znalezione w zasięgu lokalnym
f1!
    f3()
f2()

>>> f1()
99
```

Python przeszukuje zasięgi lokalne *wszystkich* instrukcji `def` zawierających funkcję, od wewnętrznej do zewnętrznej, po sprawdzeniu zasięgu lokalnego funkcji i przed sprawdzeniem zasięgu globalnego modułu. Ten rodzaj kodu raczej nie pojawia się w praktyce. Jak wiesz, w Pythonie obowiązuje zasada, że *płaska struktura jest lepsza od zagnieżdżonej* i jest ona nadal prawdziwa, nawet po wprowadzeniu zagnieżdżonych zasięgów. Z wyjątkiem pewnych specyficznych zastosowań Twoje życie i życie Twoich współpracowników stanie się lepsze, jeżeli będziesz minimalizował liczbę zagnieżdżonych definicji funkcji.

Instrukcja `nonlocal` w Pythonie 3.x

W poprzednim podrozdziale omawialiśmy sposób, w jaki funkcje zagnieżdżone mogą odwoływać się do zmiennych z zasięgu funkcji zawierającej, nawet jeżeli taka funkcja zwróciła już wynik swojego działania. Okazuje się, że począwszy od wersji 3.x (choć nie w 2.x), możemy także modyfikować takie zmienne z zasięgu funkcji zawierającej, o ile zostaną zadeklarowane w

instrukcjach `nonlocal`. Dzięki tej instrukcji zagnieżdżone instrukcje `def` mają dostęp do odczytu i zapisu zmiennych znajdujących się w funkcjach je zawierających. To sprawia, że domknięcie zakresów zagnieżdżonych stają się bardziej przydatne, zapewniając modyfikowalne informacje o stanie.

Instrukcja `nonlocal` jest bardzo podobna pod względem formy i przeznaczenia do omówionej wcześniej instrukcji `global`. Podobnie jak `global`, deklaruje ona zmienną, która zostanie zmodyfikowana w zasięgu funkcji zawierającej. W przeciwieństwie do `global` instrukcja `nonlocal` ma zastosowanie do zmiennej z zasięgu funkcji zawierającej, a nie do zasięgu modułu poza wszystkimi instrukcjami `def`. I inaczej niż w `global` — w przypadku `nonlocal` zmienne muszą w momencie deklaracji istnieć już w zasięgu funkcji zawierającej — mogą istnieć jedynie w funkcjach zawierających i nie mogą być tworzone za pomocą pierwszego przypisania w zagnieżdżonej instrukcji `def`.

Innymi słowy, instrukcja `nonlocal` zarówno pozwala na przypisywanie do zmiennych w zasięgach funkcji zawierającej, jak i ogranicza przeszukiwanie zasięgów dla zmiennych tego typu jedynie do instrukcji `def` zawierających funkcję. W rezultacie otrzymujemy bardziej bezpośrednią i niezawodną implementację zmieniającej się informacji o stanie dla programów, które nie chcą bądź nie potrzebują klas z atrybutami, dziedziczeniem i wieloma zachowaniami.

Podstawy instrukcji `nonlocal`

W Pythonie 3.x wprowadzono nową instrukcję `nonlocal`, która ma znaczenie jedynie wewnętrz funkcji:

```
def func():
    nonlocal zmienna1, zmienna2, ...
                                         # Tutaj zmienne są OK.
    >>> nonlocal X
SyntaxError: nonlocal declaration not allowed at module level
```

Powыższa instrukcja pozwala, by funkcja zagnieżdżona modyfikowała jedną lub większą liczbę zmiennych zdefiniowanych w zasięgu funkcji zawierającej. W Pythonie 2.x, kiedy instrukcja `def` jednej funkcji była zagnieżdżona w innej, funkcja zagnieżdżona mogła odwoływać się do dowolnych zmiennych zdefiniowanych przez przypisanie w zasięgu funkcji zawierającej, jednak nie mogła ich modyfikować. W wersji 3.x zadeklarowanie zmiennych z zasięgów funkcji zawierającej w instrukcji `nonlocal` pozwala funkcji zagnieżdżonej na przypisanie i tym samym również modyfikację takich zmiennych.

W ten sposób funkcje zawierające udostępniają informacje o stanie *do zapisu*, pamiętane w czasie późniejszego wywołania funkcji zagnieżdżonych. Możliwość zmiany stanu sprawia, że jest on bardziej użyteczny dla funkcji zagnieżdżonej (wyobraźmy sobie na przykład licznik w zasięgu funkcji zawierającej). W wersji 2.x programiści osiągali ten sam cel za pomocą klas lub innych metod. Ponieważ funkcje zagnieżdżone stały się teraz często wykorzystywany wzorcem kodu służącym do zachowywania stanu, instrukcja `nonlocal` sprawia, że zadanie to jest łatwiejsze do wykonania.

Poza umożliwieniem modyfikacji zmiennych w instrukcjach `def` funkcji zawierających instrukcja `nonlocal` wymusza także pewną kwestię w związku z referencjami. Podobnie jak instrukcja `global`, `nonlocal` powoduje, że wyszukiwanie zmiennej w instrukcji rozpoczyna się w zasięgu instrukcji `def` funkcji zawierających, a nie w zasięgu lokalnym dla funkcji deklarującej. Instrukcja `nonlocal` oznacza zatem: „Całkowicie pomin mój zasięg lokalny”.

Tak naprawdę zmienne wymienione w `nonlocal` w momencie dotarcia do `nonlocal` muszą być wcześniej zdefiniowane w instrukcji `def` funkcji zawierającej — inaczej zwrócony zostanie błąd. Rezultat jest podobny do `global` — instrukcja `global` oznacza, że zmienne znajdują się w module zawierającym, natomiast `nonlocal` oznacza, że znajdują się one w funkcji zawierającej.

Instrukcja `nonlocal` jest przy tym nawet bardziej rygorystyczna — przeszukiwanie zasięgów ograniczone jest tylko do funkcji zawierających. Zmienne nielokalne mogą się pojawić jedynie w instrukcjach `def` funkcji zawierających, a nie w zasięgu globalnym modułu czy we wbudowanych modułach poza instrukcjami `def`.

Dodanie `nonlocal` nie zmienia ogólnych reguł zasięgów referencji zmiennych — nadal działają one tak jak poprzednio, zgodnie z opisaną wcześniej regułą LEGB. Instrukcja `nonlocal` służy przede wszystkim do umożliwienia modyfikacji zmiennych znajdujących się w zasięgach funkcji zawierających, a nie tylko odwoływania się do nich. Zarówno `global`, jak i `nonlocal` ograniczają jednak nieco reguły wyszukiwania, kiedy używa się ich w funkcji:

- Instrukcja `global` sprawia, że przeszukiwanie zasięgów rozpoczyna się w zasięgu modułu zawierającego, i pozwala na przypisywanie zmiennych tam się znajdujących. Jeżeli zmienna nie znajduje się w module, przeszukiwanie zasięgów jest kontynuowane w zasięgu wbudowanym, jednak przypisania do zmiennych globalnych zawsze tworzą lub modyfikują je w zasięgu modułu.
- Instrukcja `nonlocal` ogranicza przeszukiwanie zasięgów do instrukcji `def` funkcji zawierających, wymaga, by zmienne już tam istniały, i pozwala na przypisywanie ich. Przeszukiwanie zasięgów nie jest kontynuowane w zasięgach globalnym czy wbudowanym.

W Pythonie 2.x referencje do zmiennych z zasięgów instrukcji `def` są dozwolone, natomiast przypisania nie. Można jednak nadal wykorzystywać klasy z jawnymi atrybutami do uzyskania tego samego efektu zmieniających się informacji o stanie jak w przypadku zmiennych nielokalnych (i w niektórych kontekstach będzie to lepsze rozwiązanie). Zmienne globalne oraz atrybuty funkcji także mogą czasami posłużyć do tych samych celów.Więcej informacji na ten temat za moment — na razie przyjrzyjmy się przykładowi kodu, by nieco skonkretyzować podane wiadomości.

Instrukcja `nonlocal` w akcji

Przejdzmy do przykładów — wszystkich wykonywanych w Pythonie 3.x. Referencje do zasięgów instrukcji `def` funkcji zawierających w wersji 3.x działają tak samo jak w wersji 2.x. W poniższym kodzie funkcja `tester` tworzy i zwraca funkcję `nested`, która będzie wywołana później, a referencja do zmiennej `state` w funkcji `nested` odwzorowana zostaje na zasięg lokalny funkcji `tester` za pomocą normalnych reguł przeszukiwania zasięgów:

```
C:\code>c:\python33\python
>>> def tester(start):
        state = start                      # Referencja do zmiennej nielokalnej
        działa normalnie
        def nested(label):                 # Pamięta stan w zasięgu funkcji
            print(label, state)           # zawierającej
            return nested
        return nested

>>> F = tester(0)
>>> F('mielonka')
mielonka 0
>>> F('szynka')
```

```
szynka 0
```

Modyfikacja zmiennej w zasięgu instrukcji `def` funkcji zawierającej nie jest jednak domyślnie dozwolona — tak samo jest również w wersji 2.x:

```
>>> def tester(start):
    state = start
    def nested(label):
        print(label, state)
        state += 1                    # Domyślnie nie może się zmienić (w 2.x
    też nie)
        return nested

>>> F = tester(0)
>>> F('mielonka')
UnboundLocalError: local variable 'state' referenced before assignment
```

Użycie zmiennych nielokalnych w celu modyfikacji

Obecnie w Pythonie 3.x, jeżeli zadeklarujemy zmienną `state` w zasięgu funkcji `tester` jako `nonlocal` wewnątrz funkcji `nested`, możemy ją zmodyfikować również wewnątrz funkcji zagnieżdzonej. Takie rozwiązanie działa, mimo że funkcja `tester` zwróciła wartość i zakończyła działanie, zanim wywołaliśmy funkcję `nested` za pośrednictwem zmiennej `F`:

```
>>> def tester(start):
    state = start                  # Każde wywołanie otrzymuje własną
    zmienią state
    def nested(label):
        nonlocal state            # Pamięta state z zasięgu funkcji
        zawierającej
        print(label, state)
        state += 1                 # Można zmienić, jeśli nonlocal
    return nested

>>> F = tester(0)
>>> F('mielonka')                # Inkrementuje state z każdym
wywołaniem
mielonka 0
>>> F('szynka')
szynka 1
>>> F('jajka')
jajka 2
```

Jak zwykle w przypadku referencji do zasięgów funkcji zawierającej — funkcję fabryczną tester możemy wywołać kilka razy w celu otrzymania kilku kopii jej stanu w pamięci. Obiekt state w zasięgu funkcji zawierającej jest dołączony do zwracanego obiektu funkcji nested. Każde wywołanie tworzy nowy, odrębny obiekt state w taki sposób, że uaktualnienie stanu jednej funkcji nie będzie miało wpływu na inną. Poniższy kod stanowi kontynuację poprzedniej interakcji:

```
>>> G = tester(42)                      # Utworzenie nowej funkcji tester
rozpoczynającej się od 42
>>> G('mielonka')
mielonka 42
>>> G('jajka')                         # Informacje o stanie uaktualnione do 43
jajka 43
>>> F('bekon')                          # Dla F pozostają takie, jakie były: 3
bekon 3                                  # Każde wywołanie ma inne informacje o
                                         stanie
```

W tym sensie zmienne nonlocal Pythona są bardziej funkcjonalne niż zmienne lokalne funkcji typowo używane w wielu innych językach programowania: w takiej funkcji zmienne nonlocal w każdym wywołaniu posiadają inne wartości reprezentujące bieżący stan.

Przypadki graniczne

Choć zmienne nonlocal w niektórych zastosowaniach są bardzo użyteczne, istnieje jednak kilka elementów, na które należy uważać. Po pierwsze w przeciwieństwie do instrukcji global zmienne nonlocal naprawdę muszą być wcześniej przypisane w zasięgu instrukcji def funkcji zawierającej, kiedy instrukcja nonlocal będzie obliczana, gdyż inaczej otrzymamy błąd — nie możemy ich tworzyć dynamicznie, przypisując na nowo w zasięgu funkcji zawierającej. W praktyce są one sprawdzane w czasie definiowania funkcji, zanim zostanie wywołana funkcja zawierająca lub zagnieźdzona:

```
>>> def tester(start):
    def nested(label):
        nonlocal state          # Zmienne nonlocal muszą już istnieć w
        zasięgu funkcji zawierającej!
        state = 0
        print(label, state)
    return nested
```

SyntaxError: no binding for nonlocal 'state' found

```
>>> def tester(start):
    def nested(label):
        global state           # Zmienne globalne nie muszą istnieć
        podczas deklarowania
        state = 0              # To polecenie tworzy teraz zmienną w
        module
        print(label, state)
```

```
        return nested

>>> F = tester(0)
>>> F('abc')
abc 0
>>> state
0
```

Po drugie instrukcja `nonlocal` ogranicza przeszukiwanie zasięgów jedynie do instrukcji `def` funkcji zawierających. Zmienne nielokalne nie są szukane w zasięgu globalnym modułu zawierającego czy zasięgu globalnym poza instrukcjami `def`, nawet jeśli tam już są:

```
>>> spam = 99
>>> def tester():
    def nested():
        nonlocal spam
        # Musi być w instrukcji def, nie
        w module!
        print('Aktualna wartość=', spam)
        spam += 1
    return nested
```

```
SyntaxError: no binding for nonlocal 'spam' found
```

Powyższe ograniczenia mają sens, jeśli sobie uświadomimy, że Python nie wiedziałby, w którym zasięgu funkcji zawierającej miałby utworzyć nową zmienną. I tak w powyższym przykładzie — czy zmienna `spam` powinna być przypisana w funkcji `tester`, czy też może w zawierającym ją module? Ponieważ nie jest to oczywiste, Python musi znać zmienne nielokalne w momencie *tworzenia* funkcji, a nie jej *wywołania*.

Czemu służą zmienne `nonlocal`? Opcje zachowania stanu

Biorąc pod uwagę dodatkowy poziom skomplikowania funkcji zagnieżdżonych, można się zastanawiać, po co to całe zamieszanie. Choć trudno to zobaczyć w prostych przykładach, informacje o stanie w wielu programach stają się kluczowe. Podczas gdy funkcje mogą zwracać wyniki swojego działania, ich zmienne lokalne zwykle nie zachowują innych wartości, które muszą być zachowywane między kolejnymi wywołaniami. Co więcej, wiele aplikacji wymaga, aby takie wartości różniły się w zależności od kontekstu użytkowania.

Jak wspominaliśmy wcześniej, istnieje wiele sposobów „zapamiętywania” informacji pomiędzy wywołaniami funkcji i metod w Pythonie. Choć każdy z nich wiąże się z pewnymi kompromisami, zmienne `nonlocal` poprawiają sytuację w przypadku referencji do zasięgów funkcji zawierających — instrukcja `nonlocal` pozwala na przechowywanie w pamięci większej liczby kopii zmieniających się stanów, a także zaspokaja proste potrzeby w zasięgu przechowywania stanu w sytuacjach, w których stosowanie klas może nie być uzasadnione, a

zmienne globalne nie mają zastosowania, chociaż atrybuty funkcji mogą często pełnić podobne role w bardziej przenośny sposób. Przyjrzyjmy się zatem dostępnym opcjom:

Zachowanie stanu: zmienne nonlocal (tylko w wersji 3.x)

Jak widzieliśmy w poprzednim podrozdziale, poniższy kod pozwala na przechowywanie stanu, a także modyfikowanie go w zasięgu funkcji zawierającej. Każde wywołanie funkcji `tester` tworzy niewielki samodzielny pakiet zmieniających się informacji, w którym zmienne nie wchodzą w konflikt z żadnymi innymi częściami programu:

```
>>> def tester(start):
    state = start
    # Każde wywołanie otrzymuje
    własną zmienną state

    def nested(label):
        nonlocal state
        # Pamięta state z zasięgu
        funkcji zawierającej
        print(label, state)
        state += 1
        # Można zmienić, jeśli
        nonlocal
        return nested

>>> F = tester(0)
>>> F('mielonka')
# Stan widziany tylko w
zasięgu funkcji zawierającej
spam 0
>>> F.state
AttributeError: 'function' object has no attribute 'state'
```

Musimy zadeklarować zmienne jako nielokalne tylko wtedy, gdy muszą zostać zmienione (inne odniesienia do zakresu obejmującego są automatycznie zachowywane jak zwykle), a nazwy nielokalne są nadal niewidoczne poza funkcją zawierającą.

Niestety, powyższy kod działa jedynie w Pythonie 3.x. Dla osób pracujących w Pythonie 2.x w zależności od potrzeb dostępne są inne opcje. W kolejnych trzech podrozdziałach prezentujemy pewne alternatywy. Część kodu prezentowanego w tych sekcjach używa narzędzi, których jeszcze nie omawialiśmy, i częściowo może spełniać rolę podglądu, ale będziemy się starać utrzymać przykłady w prostocie, aby można było je ze sobą porównywać i analizować.

Zachowanie stanu: zmienne globalne – tylko jedna kopia

Jednym z często spotykanych sposobów na uzyskanie efektu instrukcji `nonlocal` w Pythonie 2.x i wcześniejszych wersjach jest po prostu przeniesienie stanu do *zasięgu globalnego* (modułu zawierającego):

```
>>> def tester(start):
```

```

global state                                # Przeniesienie do modułu w celu
modyfikacji

state = start                            # zmienne global pozwalają na dokonywanie
zmian w zasięgu modułu

def nested(label):

    global state
    print(label, state)
    state += 1
return nested

>>> F = tester(0)
>>> F('mielonka')                      # Każde wywołanie inkrementuje
współdzieloną zmienną globalną state
mielonka 0
>>> F('jajka')
jajka 1

```

Takie rozwiązanie w tym przypadku działa, jednak wymaga deklaracji `global` w obu funkcjach i jest podatne na konflikty nazw zmiennych w zasięgu globalnym (co będzie, jeśli nazwa zmiennej `state` jest już wykorzystywana?). Poważniejszym i bardziej subtelnym problemem jest to, że rozwiązanie to pozwala na istnienie tylko *jednej współdzielonej kopii* informacji o stanie w zasięgu modułu. Jeżeli znowu wywołamy funkcję `tester`, przywrócimy zmienną `state` modułu do początkowej wartości, a poprzednie wywołania zobaczą, że ich zmienna `state` została nadpisana:

```

>>> G = tester(42)                      # Przywraca wartość jedynej kopii
zmiennej state w zasięgu globalnym

>>> G('tost')
tost 42
>>> G('bekon')
bekon 43
>>> F('szynka')                        # 0 rany, mój licznik został nadpisany!
szynka 44

```

Jak pokazano wcześniej, przy użyciu zmiennych `nonlocal` w miejsce zmiennych globalnych, każde wywołanie funkcji `tester` zapamiętuje własną, unikalną kopię obiektu `state`.

Zachowanie stanu: klasy — jawne atrybuty (wprowadzenie)

Inną receptą na uzyskanie modyfikowalnych informacji o stanie w Pythonie 2.x oraz wcześniejszych wersjach jest wykorzystanie *klas z atrybutami*, dzięki czemu dostęp do informacji o stanie odbywa się w sposób bardziej jawnym niż w przypadku niejawnej magii reguł przeszukiwania zasięgów. Dodatkową zaletą jest to, że każda instancja klasy otrzymuje świeżą

kopię informacji o stanie, będącą naturalnym produktem ubocznym modelu obiektów Pythona. Klasy obsługują również dziedziczenie i wiele zachowań, a także mają różne inne narzędzia.

Nie omawialiśmy jeszcze klas zbyt szczegółowo, jednak tytułem wstępu poniżej zamieszczały wcześniejszy wykorzystane funkcje `tester` i `nested` utworzone jako klasy. Stan jest zapisywany w obiektach w sposób jawnego w miarę ich tworzenia. By poniższy kod miał sens, musimy wiedzieć, że instrukcja `def` wewnętrznych `class` działa dokładnie tak samo jak instrukcja `def` poza `class`, z wyjątkiem faktu, iż argument `self` funkcji automatycznie otrzymuje domniemany podmiot wywołania (obiekt instancji utworzony przez wywołanie samej klasy). Funkcja o nazwie `__init__` jest uruchamiana automatycznie po wywołaniu klasy:

```
>>> class tester:                                     # Alternatywna oparta na klasach (patrz
    część VI)

        def __init__(self, start):                  # Przy tworzeniu obiektu stan jest
            self.state = start                      # ...zapisywany w nowym obiekcie w
            sposób jawnego

        def nested(self, label):                   # Jawną referencję do zmiennej state
            print(label, self.state)               # Modyfikacja jest zawsze dozwolona

            self.state += 1

>>> F = tester(0)                                    # Tworzenie instancji, wywołanie
__init__

>>> F.nested('mielonka')                         # F przekazywane jest do self
mielonka 0

>>> F.nested('szynka')
szynka 1
```

W klasach zapisujemy *każdy* atrybut jawnie, niezależnie od tego, czy został zmieniony, czy tylko pojawiło się do niego odwołanie. Atrybuty są dostępne poza klasą. Jeżeli chodzi o funkcje zagnieżdżone i zmienne `nonlocal`, alternatywne rozwiązywanie z klasami pozwala na zapamiętanie wielu kopii danych:

```
>>> G = tester(42)                                # Każda instancja otrzymuje nową kopię
zmiennej state

>>> G.nested('tost')                            # Zmiana jednej nie wpływa na
pozostałe

tost 42

>>> G.nested('bekon')
bekon 43

>>> F.nested('jajka')                           # Zmienna state dla F jest taka, na
jakiej skończyliśmy

jajka 2

>>> F.state                                     # Dostęp do state poza klasą
```

Dzięki dodatkowej odrobinie magii, którą wyjaśnimy w dalszej części książki, możemy także za pomocą przeciążania operatorów sprawić, że nasza klasa będzie przypominała wywoływalną funkcję. Metoda `__call__` przechwytuje bezpośrednie wywołania dla instancji, dzięki czemu nie musimy wywoływać nazwanej metody:

```
>>> class tester:
    def __init__(self, start):
        self.state = start
    def __call__(self, label):      # Przechwycenie bezpośrednich wywołań
        print(label, self.state)    # .nested() nie jest wymagane
        self.state += 1

>>> H = tester(99)
>>> H('sok')                  # Wywołanie metody __call__
sok 99
>>> H('naleśniki')
naleśniki 100
```

Nie ma co zajmować się szczegółami powyższego kodu na tym etapie książki — klasy omówimy dogłębnie w części VI, a narzędziom służącym do przeciążania operatorów, takim jak `__call__`, przyjrzymy się w rozdziale 30., dlatego warto zachować ten kod na później. Najważniejsze jest to, że klasy mogą sprawić, iż informacje o stanie staną się bardziej oczywiste, wykorzystując do tego jawnie przypisywanie atrybutów w miejscu przeszukiwania zasięgów. Ponadto atrybuty klas są zawsze modyfikowalne i nie wymagają zmiennych `nonlocal`, a klasy są zaprojektowane tak, aby można je było łatwo skalować do implementacji bogatszych obiektów o wielu atrybutach i zachowaniach.

Choć wykorzystywanie klas na potrzeby informacji o stanie jest z reguły dobrym zwyczajem, w klasach tego typu, gdzie stan to pojedynczy licznik, może to być *przesadą*. Tak trywialne przypadki stanu są o wiele bardziej powszechnne, niż mogłoby się wydawać. W takich kontekstach zagnieżdżone instrukcje `def` są często lżejszym rozwiązaniem od tworzenia klas, zwłaszcza w przypadku osób niezaznajomionych z programowaniem zorientowanym obiektywem. Istnieją także sytuacje, w których zagnieżdżone instrukcje `def` mogą wręcz działać lepiej od klas (przykład znacznie wykraczający poza zakres tego rozdziału znajduje się w rozdziale 39., przy okazji omawiania *dekoratorów metod*).

Zachowanie stanu: atrybuty funkcji (w wersjach 3.x i 2.x)

W tej ostatniej, bardziej przenośnej i często prostszej do zaimplementowania opcji zachowania stanu możemy czasami osiągnąć ten sam efekt, korzystając z *atributów funkcji* — zmiennych zdefiniowanych przez użytkownika i dołączanych bezpośrednio do funkcji. Kiedy dołączasz atrybuty zdefiniowane przez użytkownika do funkcji zagnieżdżonych generowanych przez zawierające funkcje fabrykujące, mogą one również służyć do przechowywania w każdym wywołaniu modyfikowalnych informacji o stanie, podobnie jak to miało miejsce w przypadku zmiennych `nonlocal`. Takie zdefiniowane przez użytkownika nazwy atrybutów nie będą kolidować z nazwami, które Python sam tworzy, a nazwy `nonlocal` powinny być używane tylko

dla zmiennych stanów, które *muszą być zmieniane*; inne odwołania do zasięgów zostają zachowane i działają normalnie.

Co najważniejsze, ten schemat postępowania jest *przenośny* — podobnie jak klasy — ale w przeciwnieństwie do zmiennych nonlocal atrybuty funkcji działają zarówno w Pythonie 3.x, jak i 2.x. W rzeczywistości są one dostępne od wersji 2.1, znacznie dłużej niż zmienne nonlocal w wersji 3.x. Ponieważ funkcje fabrykujące i tak wywołują nową funkcję przy każdym wywołaniu, nie wymaga to dodatkowych obiektów — atrybuty nowej funkcji stają się stanem dla danego wywołania w bardzo podobny sposób jak zmienne nonlocal i w podobny sposób są powiązane z generowaną funkcją w pamięci.

Ponadto atrybuty funkcji umożliwiają dostęp do zmiennych stanu *poza* funkcją zagnieżdżoną, takich jak atrybuty klas; w przypadku zmiennych nonlocal wartości stanu można zobaczyć bezpośrednio tylko w zagnieżdżonej definicji def. Jeżeli chcesz uzyskać dostęp do licznika wywołań z zewnętrz, w tym modelu możesz skorzystać z prostej funkcji pobierającej wartości atrybutów.

Oto ostateczna wersja naszego przykładu opartego na tej technice — zastępuje ona zmienną nonlocal atrybutem dołączonym do zagnieżdżonej funkcji. Ten schemat na pierwszy rzut oka może wydawać się nieintuicyjny; uzyskujesz dostęp do stanu poprzez nazwę funkcji zamiast za pomocą prostych zmiennych. Jest to jednak rozwiążanie o wiele bardziej przenośne i pozwala na dostęp do zmiennej stanu *spoza* funkcji zagnieżdżonej (przy użyciu zmiennych nielokalnych możemy jedynie zobaczyć zmienne stanu wewnątrz zagnieżdżonej instrukcji def):

```
>>> def tester(start):
    def nested(label):
        print(label, nested.state)          # nested jest w zasięgu
        funkcji zawierającej
        nested.state += 1                  # Zmiana atrybutu, a nie samej
        nested                           # Początkowy stan po
        nested.state = start             zdefiniowaniu funkcji
        return nested

>>> F = tester(0)
>>> F('mielonka')                   # F to nested z dołączonym
stanem
mielonka 0
>>> F('szynka')
szynka 1
>>> F.state                         # Można także uzyskać dostęp
do stanu spoza funkcji
2
```

Ponieważ każde wywołanie funkcji zewnętrznej powoduje powstanie nowego, zagnieżdżonego obiektu funkcji, ten schemat obsługuje wiele kopii danych modyfikowanych w każdym wywołaniu, podobnie jak to miało miejsce w przypadku zmiennych nonlocal i klas — jest to rozwiążanie, którego nie mogą zapewnić zmienne globalne:

```

>>> G = tester(42)                                # G ma własny stan, nie
nadpisuje stanu F

>>> G('jajka')
jajka 42

>>> F('szynka')
szynka 2

>>> F.state                                     # Stan jest dostępny dla
każdego wywołania

3

>>> G.state

43

>>> F is G                                      # Różne obiekty funkcji

False

```

Powyższy kod oparty jest na fakcie, że nazwa `nested` jest zmienną lokalną w zasięgu funkcji `tester` zawierającej funkcję `nested`. Tym samym można się do niej swobodnie odwoływać wewnętrz `nested`. Kod ten polega także na tym, że modyfikacja obiektu w miejscu nie jest przypisaniem do zmiennej. Kiedy inkrementowane jest `nested.state`, modyfikowana jest część obiektu, do którego odwołuje się `nested`, a nie sama zmienna `nested`. Ponieważ tak naprawdę nie przypisujemy zmiennej w zasięgu funkcji zawierającej, nie potrzebujemy instrukcji `nonlocal`.

Atrybuty funkcji są obsługiwane zarówno w Pythonie 3.x, jak i 2.x; omówimy je dalej w rozdziale 19. Co ważne, zobaczymy, że Python stosuje konwencje nazewnictwa w wersjach 2.x i 3.x, które zapewniają, że dowolne nazwy przypisywane jako atrybuty funkcji nie będą kolidować z nazwami związanymi z implementacją wewnętrzną, dzięki czemu przestrzeń nazw jest równoważna zasięgowi. Pomijając czynniki subiektywne, użyteczność atrybutów funkcji pokrywa się ze zmiennymi `nonlocal` w wersji 3.x, co sprawia, że takie zmienne stają się technicznie zbędne i znacznie mniej przenośne.

Zachowanie stanu: obiekty mutowalne — duchy przeszłości języka Python?

Warto zauważyć, że w wersjach 2.x i 3.x możemy również zmieniać obiekty *mutowalne* w zasięgu obejmującym bez deklarowania ich nazw jako `nonlocal`. Na przykład kod przedstawiony poniżej działa tak samo jak poprzednia wersja, jest również przenośny i zapewnia zmienny stan poszczególnych wywołań:

```

def tester(start):
    def nested(label):
        print(label, state[0])                      # Wykorzystujemy zmianę obiektu
        mutowalnego w miejscu
        state[0] += 1                               # Ekstraskładnia czy jakieś
        czary?
        state = [start]
    return nested

```

Przedstawiony kod wykorzystuje mutowalność list i podobnie jak w przypadku atrybutów funkcji, opiera się na fakcie, że zmiany obiektów w miejscu nie klasyfikują nazwy jako lokalnej.

Jest to być może bardziej niejasne niż atrybuty funkcji czy zmienne `nonlocal` w wersji 3.x — technika, która dystansuje nawet atrybuty funkcji, i wydaje się, że placuje się gdzieś pomiędzy sprytnym wykorzystaniem niuansów Pythona a czarną magią! Prawdopodobnie lepiej jest użyć w ten sposób nazwanych atrybutów funkcji niż list i przesunięć numerycznych, chociaż może to pojawić się w kodzie, którego będziesz musiał użyć.

Podsumowując, zmienne globalne, nielokalne, klasy oraz atrybuty funkcji oferują możliwość przechowania stanu. Zmienne globalne obsługują jedynie dane współdzielone, zmienne `nonlocal` mogą być modyfikowane tylko w wersji 3.x, klasy wymagają podstawowej znajomości programowania zorientowanego obiektowo, a zarówno klasy, jak i atrybuty funkcji pozwalają na dostęp do stanu spoza samej funkcji zagnieżdzonej. Jak zawsze wybór najlepszego narzędzia na potrzeby programu uzależniony jest od jego przeznaczenia.

W rozdziale 39. powrócimy do wszystkich opcji stanu wprowadzonych tutaj, ale w bardziej realistycznym kontekście dekoratorów, czyli narzędzi, które z natury wymagają wielopoziomowego zachowania stanu. Opcje stanu mają dodatkowe czynniki wyboru (np. wydajność), które na razie będziemy musieli pozostawić niezbadane (o sposobach mierzenia szybkości działania kodu dowiemy się w rozdziale 21.). Na razie czas przejść do omawiania trybów przekazywania argumentów.

Warto pamiętać — dostosowywanie funkcji open

W innym przykładzie rozważmy zmianę wbudowanej funkcji `open` na wersję niestandardową, zgodnie z sugestią zawartą w ramce „Jak wywrócić świat do góry nogami w Pythonie 2.x” we wcześniejszej części tego rozdziału. Jeżeli niestandardowa wersja funkcji będzie musiała wywoływać oryginał, powinieneś go zapisać przed zmianą i zachować do późniejszego użytku — to klasyczny scenariusz zachowania stanu. Co więcej, jeżeli chcesz obsługiwać wiele dostosowań tej samej funkcji, zmienne globalne tego nie załatwiają: będą Ci potrzebne stany poszczególnych dostosowań.

Poniższy kod, napisany dla Pythona 3.x i umieszczyony w pliku `makeopen.py`, jest jednym ze sposobów na osiągnięcie naszego celu (w wersji 2.x zmień wbudowaną nazwę zasięgu i sposób wyświetlania). W przykładzie używamy zagnieżdżonych zasięgów do zapamiętania wartości stanu do późniejszego użycia, bez polegania na zmiennych globalnych, które mogą powodować kolizje, i bez użycia klas, które mogą wymagać napisania większej ilości kodu, niż jest to tutaj uzasadnione:

```
import builtins

def makeopen(id):
    original = builtins.open

    def custom(*pargs, **kargs):
        print('Niestandardowe wywołanie funkcji open %r:' % id, pargs,
kargs)
        return original(*pargs, **kargs)

    builtins.open = custom
```

Aby zmienić funkcję `open` dla każdego modułu w procesie, nasz kod ponownie przypisuje ją we wbudowanym zasięgu do niestandardowej wersji zakodowanej za pomocą zagnieżdżonej instrukcji `def` po zapisaniu oryginału w zakresie zawierającym, dzięki czemu dostosowanie może go później wywołać. Kod zawiera również nowe elementy; opiera się na argumentach z gwiazdkami, które pozwalają na gromadzenie, a następnie rozpakowywanie dowolnych argumentów pozycyjnych i słów kluczowych — to zagadnienia, które tak na dobre pojawią się dopiero w następnym rozdziale. Jednak dużą część magii stanowią tutaj zagnieżdżone zamknięcia zasięgów: niestandardowa funkcja `open` znaleziona przez reguły wyszukiwania zasięgów zachowuje oryginał do późniejszego wykorzystania:

```

>>> F = open('script2.py')                      # Wywołanie wbudowanej funkcji open
z modułu builtins

>>> F.read()

import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

>>> from makeopen import makeopen      # Importowanie funkcji spełniającej
rolę resettera funkcji open

>>> makeopen('spam')                  # Niestandardowa funkcja open
wywołuje wbudowaną f. open

>>> F = open('script2.py')          # Wywołanie niestandardowej funkcji
open z modułu builtins

Custom open call 'spam': ('script2.py',) {}

>>> F.read()

import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

```

Ponieważ każde dostosowanie pamięta poprzednią wbudowaną wersję zasięgu we własnym zasięgu obejmującym, można je nawet całkiem naturalnie *zagieździć* w sposób, który nie jest obsługiwany przez zmienne globalne — każde wywołanie funkcji makeopen zapamiętuje własne wersje `id` i `original`, dzięki czemu można uruchomić wiele dostosowań:

```

>>> makeopen('eggs')                # Zagieźdżenia dostosowań również
działają!

>>> F = open('script2.py')          # Dzieje się tak, ponieważ każde
zachowuje swój stan

Custom open call 'eggs': ('script2.py',) {}

Custom open call 'spam': ('script2.py',) {}

>>> F.read()

import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

```

W tej chwili nasza funkcja po prostu dodaje zagieździone śledzenie wywołań do funkcji wbudowanej, ale ogólnie pokazana technika może mieć inne zastosowania. Oparty na klasach ekwiwalent tego rozwiązania może wymagać napisania większej ilości kodu, ponieważ musiałby jawnie zapisać wartości zmiennych `id` i `original` w atrybutach obiektowych — ale realizacja tego wymaga nieco większej wiedzy, niż zdobyliśmy dotychczas, więc potraktuj poniższy kod jako malutkie wprowadzenie do części VI:

```

import builtins

class makeopen:                      # Zobacz część VI książki: wywołanie
self()

def __init__(self, id):
    self.id = id
    self.original = builtins.open
    builtins.open = self

def __call__(self, *pargs, **kargs):
    print('Custom open call %r:' % self.id, pargs, kargs)

```

```
    return self.original(*pargs, **kargs)
```

Należy tutaj zauważać, że klasy mogą być bardziej wyraźne, ale mogą również wymagać dodatkowego kodu, gdy jedynym celem jest zachowanie stanu. Później zobaczymy dodatkowe przypadki użycia domknięcia, szczególnie podczas omawiania dekoratorów w rozdziale 39., gdzie przekonasz się, że w porównaniu z klasami w niektórych zastosowaniach praktycznych domknięcie są rozwiązaniem preferowanym.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy jedną z dwóch kluczowych koncepcji związanych z funkcjami — *zasięgi* (sposób wyszukiwania zmiennych, kiedy są one wykorzystywane). Jak już wiemy, zmienne są uznawane za lokalne dla definicji funkcji, w której zostały przypisane, o ile nie zostały specjalnie zadeklarowane jako globalne lub nielokalne. Omówiliśmy również kilka bardziej zaawansowanych koncepcji związanych z zasięgami, w tym zasięgi funkcji zagnieżdżonych oraz atrybuty funkcji. Wreszcie przyjrzaliśmy się kilku ogólnym koncepcjom związanym z zasięgami, takim jak unikanie zmiennych globalnych i modyfikacji pomiędzy plikami.

W kolejnym rozdziale będziemy kontynuować omówienie funkcji, zgłębiając drugą kluczową koncepcję z nimi związaną — przekazywanie argumentów. Jak zobaczymy, argumenty przekazywane są do funkcji przez przypisanie, jednak Python udostępnia także narzędzia pozwalające na pewną elastyczność w zakresie przekazywania elementów. Przed przejściem do tych zagadnień czas zająć się quizem sprawdzającym kwestie związane z zasięgami omówione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Jaki będzie wynik działania poniższego kodu i dlaczego?

```
>>> X = 'Mielonka'  
>>> def func():  
    print(X)
```

```
>>> func()
```

2. Jaki będzie wynik działania poniższego kodu i dlaczego?

```
>>> X = 'Mielonka'  
>>> def func():  
    X = 'NI!'  
>>> func()  
>>> print(X)
```

3. Co wyświetla poniższy kod i dlaczego?

```
>>> X = 'Mielonka'
```

```
>>> def func():
    X = 'NI'
    print(X)
```

```
>>> func()
>>> print(X)
```

4. Jakie dane wyjściowe zwraca poniższy kod i dlaczego?

```
>>> X = 'Mielonka'
>>> def func():
    global X
    X = 'NI'
```

```
>>> func()
>>> print(X)
```

5. A co z tym kodem? Jaki będzie wynik jego działania i dlaczego?

```
>>> X = 'Mielonka'
>>> def func():
    X = 'NI'
    def nested():
        print(X)
    nested()
```

```
>>> func()
>>> X
```

6. Co z tym przykładem? Jaki będzie wynik jego działania w Pythonie 3.x i dlaczego?

```
>>> def func():
    X = 'NI'
    def nested():
        nonlocal X
        X = 'Mielonka'
        nested()
    print(X)

>>> func()
```

- Podaj trzy lub większą liczbę sposobów przechowania informacji o stanie w funkcji Pythona.

Sprawdź swoją wiedzę – odpowiedzi

- Odpowiedź brzmi: 'Mielonka', ponieważ funkcja odnosi się do zmiennej globalnej z modułu ją zawierającego (ponieważ zmienna nie jest przypisana wewnątrz funkcji, uznawana jest za globalną).
- Odpowiedź znowu brzmi: 'Mielonka', ponieważ przypisanie zmiennej wewnątrz funkcji sprawia, że jest ona lokalna i w rezultacie ukrywa zmienną globalną o tej samej nazwie. Instrukcja `print` odnajduje niezmodyfikowaną zmienną w zasięgu globalnym (modułu).
- Wyświetla 'NI' w pierwszym wierszu, a 'Mielonka' w drugim, ponieważ referencja do zmiennej wewnątrz funkcji odnajduje przypisaną zmienną lokalną, natomiast referencja z `print` odnajduje zmienną globalną.
- Tym razem wyświetla tylko 'NI', ponieważ deklaracja globalna sprawia, że zmienna przypisana wewnątrz funkcji odnosi się do zmiennej z zawierającego tę funkcję zasięgu globalnego.
- Wynikiem znowu będzie 'NI' w pierwszym wierszu i 'Mielonka' w drugim, ponieważ instrukcja `print` z zagnieżdzonej funkcji odnajduje nazwę w lokalnym zasięgu funkcji zawierającej, natomiast instrukcja `print` na końcu odnajduje zmienną w zasięgu globalnym.
- Przykład ten wyświetla 'Mielonka', ponieważ instrukcja `nonlocal` (dostępna w Pythonie 3.x, ale nie w 2.x) powoduje, że przypisanie do `X` wewnątrz funkcji zagnieżdzonej modyfikuje `X` w zasięgu lokalnym funkcji zawierającej. Bez tej instrukcji przypisanie to zaklasyfikowałoby `X` jako zmienną lokalną dla funkcji zagnieżdzonej, przez co byłaby to inna zmienna. Kod wyświetliłby wtedy 'NI'.
- Choć wartości zmiennych lokalnych znikają, kiedy funkcja zwraca wartość, informacje o stanie można w funkcji Pythona zachować za pomocą współdzielonych zmiennych globalnych, referencji do zasięgów funkcji zawierających wewnątrz funkcji zagnieżdżonych lub domyślnych wartości argumentów. Atrybuty funkcji pozwalają czasami dołączać stan do samej funkcji, zamiast wyszukiwać go w zasięgach. Alternatywa polegająca na zastosowaniu programowania orientowanego obiektywnego i klas obsługuje czasami zachowanie informacji o stanie w lepszy sposób od wymienionych technik opartych na zasięgach, ponieważ odbywa się to w sposób jawnny za pomocą przypisania atrybutów; opcję tę omówimy w szóstej części książki.

[1] W pierwszym wydaniu tej książki reguła wyszukiwania zasięgów nosiła nazwę „reguły LGB”. Warstwa instrukcji `def` została w Pythonie dodana później, aby uniknąć konieczności jawnego przekazywania nazw zakresów z domyślnymi argumentami — jest to jednak zagadnienie o marginalnym znaczeniu dla początkujących użytkowników Pythona, zatem spokojnie możemy odłożyć jego omówienie do dalszej części rozdziału. Ponieważ zasięg nielokalny został rozwiązyany w Pythonie 3.x za pomocą instrukcji `nonlocal`, reguła wyszukiwania mogłaby teraz nosić nazwę LNGB, ale na szczęście zachowanie zgodności z poprzednimi wersjami ma znaczenie także w książkach! Obecna forma tego akronimu nie uwzględnia również nowszych, niejasnych zasięgów niektórych składanych komponentów i procedur obsługi wyjątków, ale tworzenie akronimów dłuższych niż cztery litery zaczyna pomału mijać się z celem!

[2] Wielowątkowość powoduje wykonywanie wywołań funkcji równolegle z resztą programu i obsługiwana jest przez moduły `_thread`, `threading` oraz `queue` biblioteki standardowej Pythona (w wersji 2.x `thread`, `threading` oraz `Queue`). Ponieważ wszystkie funkcje wątkowane wykonywane są w tym samym procesie, zasięgi globalne często służą jako jeden z rodzajów pamięci współdzielonej między nimi (wątki mogą mieć nazwy w zasięgu globalnym, a także obiekty w przestrzeni pamięci procesu). Użycie wątków jest powszechnie wykorzystywane na potrzeby długo trwających zadań w graficznych interfejsach użytkownika, do implementowania nieblokujących operacji, a także maksymalnego wykorzystania zasobów procesora. Omawianie wątków zdecydowanie wykracza poza tematykę niniejszej książki; więcej informacji na ich temat możesz znaleźć w dokumentacji biblioteki Pythona oraz w wielu pozycjach będących jej uzupełnieniem, wymienionych w „Przedmowie” (takich jak *Programming Python*).

[3] W części poświęconej pułapkom związanym z funkcjami, w rozdziale 21., zobaczymy, że używanie obiektów mutowalnych, takich jak listy i słowniki, w postaci argumentów domyślnych (na przykład `def f(a=[])`) może być problematyczne — ponieważ wartości domyślne implementowane są jako pojedyncze obiekty dołączane do funkcji, mutowalne wartości domyślne zachowują stan pomiędzy wywołaniami, a nie są inicjalizowane od nowa z każdym wywołaniem. W zależności od tego, kogo o to zapytamy, jest to uznawane albo za opcję umożliwiającą przechowanie stanu, albo za dziwną pułapkę języka programowania. Więcej szczegółowych informacji na ten temat znajdziesz na końcu rozdziału 21.

Rozdział 18. Argumenty

W rozdziale 17. omówiliśmy szczegóły dotyczące *zasięgów Pythona* — miejsc, w których zmienne są definiowane i wyszukiwane. Wiemy już, że miejsce zdefiniowania zmiennej w kodzie określa w dużej mierze jej znaczenie. W niniejszym rozdziale kontynuujemy omawianie funkcji, przechodząc do koncepcji *przekazywania argumentów* w Pythonie, czyli sposobu, w jaki obiekty przesyłane są do funkcji w charakterze danych wejściowych. Jak się niebawem przekonamy, argumenty (inaczej nazywane parametrami) przypisywane są do zmiennych w funkcji, jednak mają więcej wspólnego z referencjami do obiektów niż z zasięgami zmiennych. Zobaczmy także, że Python udostępnia dodatkowe narzędzia, takie jak słowa kluczowe, wartości domyślne oraz kolektory i ekstraktory dowolnych argumentów, zapewniające większą swobodę w przesyłaniu argumentów do funkcji, co postaramy się zilustrować za pomocą przykładów.

Podstawy przekazywania argumentów

Wcześniej w tej części książki napisałem, że argumenty przekazywane są przez *przypisanie*. Ma to kilka konsekwencji, które nie zawsze są oczywiste dla osób początkujących, a które wyjaśnię w niniejszym podrozdziale. Poniżej znajduje się przegląd najważniejszych kwestii związanych z przekazywaniem argumentów do funkcji.

- **Argumenty przekazywane są przez automatyczne przypisanie obiektów do nazw zmiennych lokalnych.** Argumenty funkcji — referencje do (potencjalnie) współdzielonych obiektów, do których odnosi się wywołujący — są kolejnym przykładem działania przypisania w Pythonie. Ponieważ referencje implementowane są jako wskaźniki, wszystkie argumenty są w rezultacie przekazywane za pomocą wskaźników. Obiekty przekazane jako argumenty nigdy nie są automatycznie kopowane.
- **Przypisanie do nazw argumentów wewnętrz funkcji nie wpływa na wywołującego.** Nazwy argumentów w nagłówku funkcji stają się w czasie wykonywania funkcji nowymi zmiennymi lokalnymi w zasięgu tej funkcji. Nazwy argumentów funkcji i nazwy zmiennych w zasięgu wywołującego nie są aliasami.
- **Modyfikacja mutowalnego argumentu w funkcji może mieć wpływ na wywołującego.** Z drugiej strony, ponieważ argumenty są po prostu przypisywane do przekazywanych obiektów, funkcje mogą modyfikować w miejscu przekazywane obiekty typu zmiennego, a wynik może wpływać na wywołującego. Argumenty zmienne mogą być danymi wejściowymi oraz wyjściowymi funkcji.

Więcej informacji na temat *referencji* można znaleźć w rozdziale 6. Wszystkie informacje z tego rozdziału mają zastosowanie również do argumentów funkcji, choć przypisanie do nazw argumentów jest automatyczne i niejawne.

Model przekazywania przez przypisanie w Pythonie nie jest do końca tym samym co opcja przekazywania argumentów przez referencje z języka C++, jednak w praktyce okazuje się bardzo zbliżony do modelu przekazywania argumentów z języka C i jemu podobnych.

- **Argumenty niemutowalne przekazywane są przez wartość.** Obiekty takie jak liczby całkowite oraz łańcuchy znaków przekazywane są przez referencję do obiektu, a nie kopowanie. Ponieważ jednak nie można zmodyfikować obiektów mutowalnych w miejscu, efekt jest taki, jakbyśmy sporządzili kopię.

- **Argumenty mutowalne przekazywane są przez wskaźnik.** Obiekty takie jak listy i słowniki są również przekazywane przez referencję do obiektu, co przypomina sposób przekazywania tablic jako wskaźników w języku C. Obiekty mutowalne mogą być modyfikowane w miejscu w funkcji, podobnie do tablic z języka C.

Oczywiście dla osób, które nigdy nie używały języka C, tryb przekazywania argumentów z Pythona będzie się wydawał jeszcze prostszy — obejmuje on po prostu przypisanie obiektów do nazw i działa tak samo bez względu na to, czy obiekty są zmienne, czy niezmienne.

Argumenty a współdzielone referencje

W celu zilustrowania działania właściwości przekazywania argumentów rozważmy poniższy kod:

```
>>> def f(a):                      # a przypisane zostaje (referencja) do
    przekazanego obiektu

    a = 99                          # Modyfikacja tylko zmiennej lokalnej a

>>> b = 88
>>> f(b)                         # a i b oba początkowo odwołują się do
tego samego obiektu 88
>>> print(b)                      # b się nie zmienia
88
```

W powyższym przykładzie w momencie wywołania funkcji za pomocą `f(b)` do zmiennej `a` przypisany zostaje obiekt 88, jednak `a` istnieje jedynie wewnątrz wywołanej funkcji. Modyfikacja `a` wewnątrz funkcji nie ma wpływu na miejsce, w którym funkcja ta jest wywołana — po prostu zmienia zmienną lokalną `a` na zupełnie inny obiekt.

To właśnie mamy na myśli, mówiąc o braku *aliasów* zmiennych — przypisanie do nazwy argumentu wewnątrz funkcji (na przykład `a=99`) nie zmienia magicznie zmiennej takiej jak `b` w zasięgu wywołania funkcji. Nazwy argumentów mogą początkowo współdzielić przekazywane obiekty (są one tak naprawdę wskaźnikami do tych obiektów), jednak tylko tymczasowo, za pierwszym wywołaniem funkcji. Po ponownym przypisaniu nazwy argumentu ten związek zanika.

Tak właśnie jest przynajmniej w przypadku przypisywania do samych *nazw* argumentów. Kiedy do argumentów przekazywane są obiekty *mutowalne*, takie jak listy i słowniki, musimy także mieć świadomość tego, że modyfikacje takich *obiektów* w miejscu mogą istnieć również po wyjściu z funkcji i tym samym wpływają na kod wywołujący. Oto przykład demonstrujący takie działanie:

```
>>> def changer(a, b):              # Do argumentów przypisano
    referencje do obiektów

    a = 2                            # Zmiana wartości jedynie
zmiennej lokalnej

    b[0] = 'mielonka'                # Zmiana współdzielonego obiektu
w miejscu

>>> x = 1
```

```

>>> L = [1, 2]                                # Wywołujący
                                                # Przekazanie obiektów
>>> changer(X, L)                            niezmennych i zmiennych
                                                # X bez zmian, L jest inne
>>> X, L                                     (1, ['mielonka', 2])

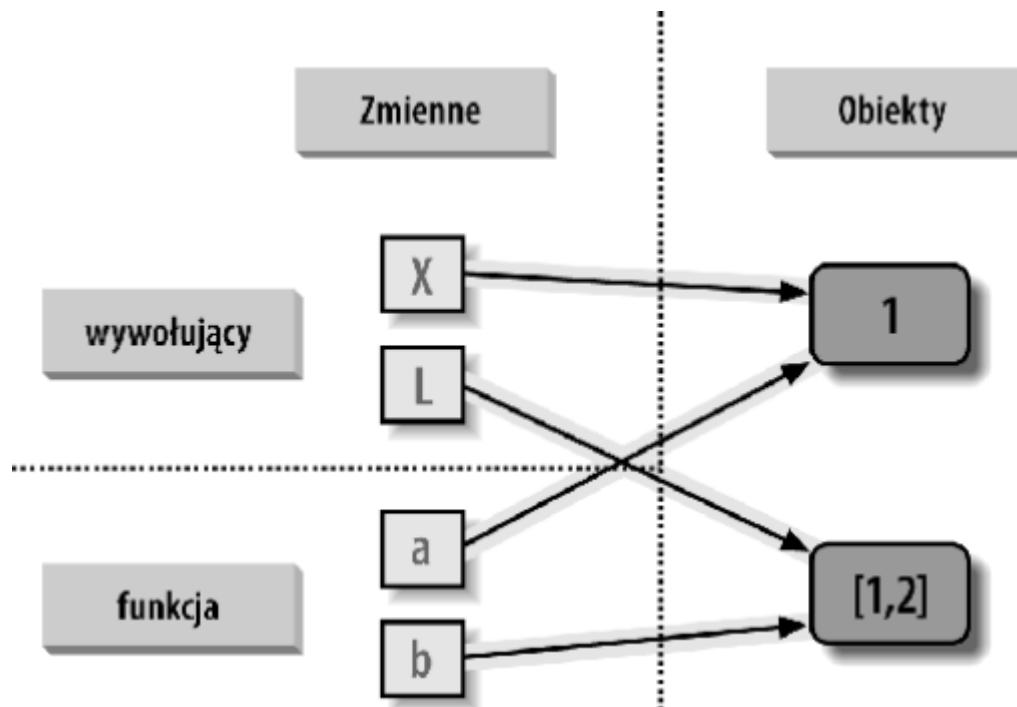
```

W powyższym kodzie funkcja `changer` przypisuje wartości do samego argumentu `a` oraz komponentu obiektu, do którego odnosi się argument `b`. Te dwa przypisania wewnętrz funkcji jedynie nieznacznie różnią się składnią, jednak dają zupełnie inne rezultaty.

- Ponieważ `a` jest zmienną lokalną w zasięgu funkcji, pierwsze przypisanie nie ma wpływu na kod wywołujący — po prostu modyfikuje ono zmienną lokalną `a`, tak by odwoływała się do zupełnie innego obiektu, i nie zmienia wiązania nazwy `X` w zasięgu wywołującym. Działa to tak samo jak w poprzednim przykładzie.
- Argument `b` również jest zmienną lokalną, jednak przekazuje się do niego obiekt zmienny (listę, do której wywołującym odwołuje się `L`). Ponieważ drugie przypisanie jest modyfikacją obiektu w miejscu, wynik przypisania do `b[0]` w funkcji ma wpływ na wartość `L` po zwróceniu tej funkcji.

Tak naprawdę druga instrukcja przypisania w funkcji `changer` nie modyfikuje `b`, zmienia jedynie część obiektu, do którego aktualnie odnosi się `b`. Ta zmiana w miejscu ma wpływ jedynie na kod wywołujący, ponieważ zmodyfikowany obiekt istnieje dłużej od wywołania funkcji. Nazwa `L` także się nie zmieniła — nadal odwołuje się do tego samego, zmodyfikowanego obiektu — jednak wydaje się, jakby `L` było inne po wywołaniu, ponieważ wartość, do której odwołuje się ta zmienna, została zmodyfikowana wewnętrz funkcji. W efekcie lista `L` służy zarówno jako wejście, jak i wyjście funkcji.

Na rysunku 18.1 zilustrowano wiązania między nazwami a obiektami istniejące tuż po wywołaniu funkcji, a przed wykonaniem jej kodu.



Rysunek 18.1. Referencje a argumenty. Ponieważ argumenty przekazywane są przez przypisanie, nazwy argumentów w funkcji mogą współdzielić obiekty ze zmiennymi z zasięgu wywołania. Z tego powodu modyfikacja zmiennych argumentów funkcji w miejscu może mieć wpływ na kod wywołujący. Na rysunku widać, że zmienne a i b z funkcji na początku są referencjami do obiektów, do których przy pierwszym wywołaniu funkcji odnoszą się zmienne X oraz L. Modyfikacja listy przez zmienną b sprawia, że po zakończeniu wywołania lista L okazuje się inną

Jeśli przykład ten nadal wydaje się niezrozumiałym, może nam pomóc wiedza, że wynik automatycznego przypisania przekazanych argumentów jest taki sam jak wykonanie serii prostych instrukcji przypisania. Jeżeli chodzi o pierwszy argument, przypisanie nie ma wpływu na wywołującego.

```
>>> X = 1
>>> a = X
# Współdzielą jeden obiekt
>>> a = 2
# Przestawia tylko 'a', 'X'
nadal jest równe 1
>>> print(X)
1
```

Przypisanie z drugiego argumentu wpływa natomiast na zmienną w wywołującym, ponieważ jest to modyfikacja obiektu w miejscu.

```
>>> L = [1, 2]
>>> b = L
# Współdzielą ten sam obiekt
>>> b[0] = 'mielonka'
# Modyfikacja w miejscu: 'L'
także widzi zmianę
>>> print(L)
['mielonka', 2]
```

Jeżeli przypomnimy sobie naszą dyskusję na temat współdzielonych obiektów mutowalnych z rozdziałów 6. i 9., rozpoznamy tutaj działanie tego samego zjawiska. Modyfikacja obiektu mutowalnego w miejscu może mieć wpływ również na inne referencje do tego obiektu. Tutaj rezultat polega na sprawieniu, by jeden z argumentów działał zarówno jako wejście, jak i wyjście funkcji.

Unikanie modyfikacji argumentów mutowalnych

Takie działanie modyfikacji argumentów mutowalnych w miejscu nie jest błędem — to po prostu sposób przekazywania argumentów w Pythonie, który okazuje się być szeroko stosowany w praktyce. Argumenty domyślnie są w Pythonie przekazywane do funkcji przez referencję (czyli wskaźnik), ponieważ takiego czegoś normalnie chcemy. Oznacza to, że możemy przekazywać w programach duże obiekty bez wykonywania przy okazji kilku ich kopii. Możemy również z łatwością po drodze uaktualnić te obiekty. Tak naprawdę, jak zobaczymy w szóstej części książki, model klas Pythona jest *uzależniony* od modyfikacji przekazanych argumentów `self` w miejscu w celu uaktualnienia stanu obiektu.

Jeżeli jednak nie życzymy sobie, by modyfikacje w miejscu wewnętrz funkcji wpływały na obiekty, które do niej przekazujemy, możemy po prostu w jawnym sposób wykonywać kopie obiektów mutowalnych, w taki sam sposób jak robiliśmy to w rozdziale 6. W przypadku argumentów funkcji możemy zawsze skopiować listę w momencie wywołania za pomocą narzędzi takich jak `list`, `list.copy` (od wersji 3.3) lub pustego wycinka:

```
L = [1, 2]
changer(X, L[:])
    # Przekazanie kopii, tak by 'L'
    się nie zmieniło
```

Możemy również dokonać kopiowania wewnątrz samej funkcji, jeżeli nigdy nie chcemy modyfikować przekazanych obiektów bez względu na sposób wywołania funkcji.

```
def changer(a, b):
    b = b[:]
    # Kopia listy w celu uniknięcia
    wpływu na wywołującego

    a = 2
    # Modyfikacja tylko kopii listy

    b[0] = 'mielonka'
```

Oba schematy kopирования nie zatrzymują modyfikacji obiektu w funkcji, a jedynie zapobiegają przeniesieniu tych zmian na wywołującego. Aby naprawdę zapobiec modyfikacjom, zawsze możemy przekształcić obiekty na niemutowalne. Na przykład próba dokonania zmian w krotkach kończy się zgłoszeniem wyjątku.

```
L = [1, 2]
changer(X, tuple(L))
    # Przekazanie krotki – zmiany są
    błędami
```

Takie rozwiązanie wykorzystuje wbudowaną funkcję `tuple`, która tworzy nową krotkę ze wszystkich elementów sekwencji (a tak naprawdę z dowolnego obiektu, na którym można wykonywać iterację). To nieco ekstremalne rozwiązanie — ponieważ wymusza napisanie funkcji w taki sposób, by nigdy nie modyfikowała ona przekazywanych argumentów, rozwiązanie to może narzucać funkcji więcej ograniczeń, niż powinno, dlatego należy go raczej unikać. Nigdy nie wiemy, kiedy modyfikacja argumentów może się w przyszłości przydać do innych wywołań. Korzystanie z tej techniki sprawi również, że funkcja straci możliwość wywoływania na argumentach metod list, w tym tych, które wcale nie modyfikują obiektu w miejscu.

Najważniejsze, co należy zapamiętać, to to, że funkcje mogą aktualniać przekazane do nich obiekty mutowalne (na przykład listy i słowniki). Nie musi to być problemem, jeżeli się tego spodziewamy, i często jest przydatne. Co więcej, funkcje modyfikujące przekazane zmienne obiekty w miejscu zostały najprawdopodobniej celowo zaprojektowane do takiego działania — zmiana ta jest częścią dobrze zdefiniowanego API, którego nie powinniśmy naruszać, wykonując kopie.

Trzeba o tym jednak pamiętać. Jeżeli jakiś obiekt niespodziewanie się nam zmienia, należy sprawdzić, czy winowającą nie jest wywołana funkcja, i w razie potrzeby wykonać kopie przekazywanych obiektów.

Symulowanie parametrów wyjścia i wielu wyników działania

Omówiliśmy już instrukcję `return` i wykorzystaliśmy ją nawet w kilku przykładach. A oto kolejny sposób wykorzystania tej instrukcji. Ponieważ instrukcja `return` może zwrócić dowolny rodzaj obiektu, może również zwracać *wiele wartości*, pakując je w krotkę czy inny typ kolekcji. Tak naprawdę, choć Python nie obsługuje czegoś, co niektóre języki programowania nazwują przekazywaniem argumentów z „wywołaniem przez referencję”, zazwyczaj możemy to symulować, zwracając krotki i przypisując wyniki z powrotem do oryginalnych zmiennych z kodu wywołującego.

```
>>> def multiple(x, y):
```

```

x = 2                                # Modyfikacja jedynie zmiennych
lokalnych

y = [3, 4]

return x, y                          # Zwrócenie wielu nowych
wartości w krotce

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)          # Przypisanie wyników do
zmiennych wywołującego
>>> X, L
(2, [3, 4])

```

Wygląda to tak, jakby kod zwracał dwie wartości, jednak tak naprawdę zwraca jedną — krotkę dwuelementową z pominiętymi opcjonalnymi nawiasami. Po zwróceniu wartości z funkcji możemy wykorzystać przypisanie krotek do rozpakowania części zwracanej krotki (jeżeli w międzyczasie zapomnisz, dlaczego takie rozwiązywanie działa, powinieneś wrócić do omówienia krotek w rozdziale 4. i 9. oraz podrozdziału „Instrukcje przypisania” z rozdziału 11). Rezultatem zastosowania takiego wzorca programistycznego będzie zarówno zwracanie wielu wyników działania, jak i symulowanie *parametrów wyjściowych* z innych języków programowania za pomocą jawnego przypisania. W tym przypadku zmienne X i L mogą się zmienić po wywołaniu, ale tylko dlatego, że jest to nakazane w kodzie.



Rozpakowywanie argumentów w Pythonie 2.x: Poprzedni przykład rozpakowuje krotkę zwróconą przez funkcję za pomocą przypisania krotki. W Pythonie 2.x można również automatycznie rozpakowywać krotki w argumentach przekazanych do funkcji. W wersji 2.x funkcja zdefiniowana za pomocą poniższego nagłówka:

```
def f((a, (b, c))):
```

może być wywołana z krotkami, które pasują do oczekiwanej struktury — `f((1, (2, 3)))` przypisuje a, b oraz c do, odpowiednio, 1, 2 i 3. Oczywiście przekazana krotka może także być obiektem utworzonym przed wywołaniem (`f(T)`). Taka składnia instrukcji `def` nie jest już obsługiwana w Pythonie 3.x. Zamiast tego funkcję należy zapisać w poniższy sposób:

```
def f(T): (a, (b, c)) = T
```

w celu rozpakowania krotki w jawniej instrukcji przypisania. Ta jawną postać działa zarówno w wersji 3.x, jak i 2.x. Rozpakowywanie argumentów jest nieco niejasną i rzadko wykorzystywaną opcją w Pythonie 2.x (ale nadal spotykana w niektórych programach). Co więcej, nagłówki funkcji w wersji 2.x obsługują jedynie przypisanie sekwencji z wykorzystaniem krotki. Bardziej uniwersalne przypisywanie sekwencji (na przykład `def f((a, [b, c])):`) zwracają błędy składni także w Pythonie 2.x i wymagają stosowania formy z jawnym przypisaniem wymaganej w wersji 3.x. I odwrotnie, dowolne sekwencje w wywołaniu z powodzeniem dopasowują krotki w nagłówku (np. `f((1, [2, 3])), f((1, "ab"))`).

Składnia argumentów z rozpakowywaniem krotek jest również zabroniona w wersji 3.x w listach argumentów funkcji `lambda`; przykład rozpakowywania argumentów funkcji `lambda` znajdziesz w ramce „Warto pamiętać: listy składane i funkcja `map`” w rozdziale 20. Nieco asymetrycznie, przypisywanie

rozpakowywania krotek jest nadal automatyczne w pętlach `for` Pythona 3.x; przykłady takiego działania można znaleźć w rozdziale 13.

Specjalne tryby dopasowywania argumentów

Jak widzieliśmy przed chwilą, argumenty są w Pythonie zawsze przekazywane przez *przypisanie*. Nazwy z nagłówka `def` są przypisywane do przekazywanych obiektów. Na bazie tego modelu Python udostępnia dodatkowe narzędzia modyfikujące sposób *dopasowywania* obiektów argumentów z wywołania do nazw argumentów z nagłówka przed przypisaniem. Wszystkie te narzędzia są opcjonalne, jednak pozwalają na tworzenie funkcji obsługujących bardziej elastyczne wzorce wywołania. Możliwe jest także napotkanie ich w niektórych bibliotekach.

Domyślnie argumenty dopasowywane są za pomocą *pozycji*, od lewej do prawej strony, i musimy przekazać dokładnie taką samą liczbę argumentów co nazw argumentów z nagłówka funkcji. Można jednak również określić dopasowanie po nazwie, wartościach domyślnych oraz kolektorach w celu podania dodatkowych argumentów.

Podstawy dopasowywania argumentów

Zanim zagłębimy się w szczegóły składniowe, chciałbym podkreślić, że tryby specjalne są opcjonalne i dotyczą jedynie dopasowania obiektów do nazw. Mechanizmem przekazywania po wykonaniu dopasowania nadal jest przypisanie. Tak naprawdę niektóre z tych narzędzi przeznaczone są bardziej dla osób piszących własne biblioteki niż dla programistów aplikacji. Ponieważ jednak można na nie trafić, nawet jeśli nie tworzy się ich samodzielnie, poniżej znajduje się podsumowanie dostępnych narzędzi.

Pozycyjne — dopasowanie od lewej do prawej strony

Normalny przypadek, z którego najczęściej dotychczas korzystaliśmy. Wartości przekazanych argumentów dopasowywane są do nazw argumentów w nagłówku funkcji zgodnie z pozycją, od lewej do prawej strony.

Słowa kluczowe — dopasowanie po nazwie argumentu

Kod wywołujący może również określać, który argument funkcji ma otrzymać wartość, wykorzystując w wywołaniu jego nazwę za pomocą składni `nazwa=wartość`.

Wartości domyślne — określenie wartości dla argumentów opcjonalnych, które nie zostały przekazane

Same funkcje mogą określać wartości domyślne, jakie mają otrzymać argumenty, kiedy wywołanie przekaże za mało wartości. Znów w użyciu pozostaje składnia `nazwa=wartość`.

Zmienna liczba argumentów (zbieranie) — przekazywanie dowolnej liczby argumentów zgodnie z pozycją lub słowem kluczowym

Funkcje mogą wykorzystywać argumenty specjalne poprzedzone jednym lub dwoma znakami `*` w celu zebrania dowolnej liczby dodatkowych argumentów (takie rozwiązanie określone jest często jako `varargs`, od nazwy listy argumentów o zmiennej długości w języku C); w Pythonie argumenty są gromadzone w normalnym obiekcie.

Zmienna liczba argumentów (rozpakowywanie) — przekazanie dowolnej liczby argumentów zgodnie z pozycją lub słowem kluczowym

Kod wywołujący może również użyć składni z * do rozpakowania kolekcji argumentów na pojedyncze, osobne argumenty. Jest to przeciwnieństwo użycia * w nagłówku funkcji — w nagłówku oznacza zebranie dowolnej liczby argumentów, natomiast w wywołaniu jest to rozpakowanie dowolnej liczby argumentów i przekazanie ich do funkcji jako szeregu wartości dyskretnych.

Argumenty będące słowami kluczowymi — argumenty, które muszą być przekazywane przez nazwę

W Pythonie 3.x (jednak nie w wersji 2.x) funkcje mogą określać argumenty, które muszą być przekazywane przez nazwę za pomocą argumentów ze słowami kluczowymi, a nie zgodnie z pozycją. Argumenty tego typu są zazwyczaj wykorzystywane do definiowania opcji konfiguracyjnych obok zwykłych argumentów.

Składnia dopasowania argumentów

W tabeli 18.1 przedstawiono składnię wywołującą specjalne tryby dopasowania argumentów.

Tabela 18.1. Formy dopasowywania argumentów funkcji

Składnia	Lokalizacja	Interpretacja
func(wartość)	Wywołujący	Normalny argument — dopasowanie po pozycji
func(nazwa=wartość)	Wywołujący	Słowo kluczowe — dopasowanie po nazwie
func(*obiekt_iterowalny)	Wywołujący	Przekazanie wszystkich obiektów iterowalnych jako pojedynczych argumentów pozycyjnych
func(**słownik)	Wywołujący	Przekazanie wszystkich par klucz-wartość ze słownika jako pojedynczych argumentów-słów kluczowych
def func(nazwa)	Funkcja	Normalny argument — dopasowuje przekazane wartości po pozycji lub nazwie
def func(nazwa=wartość)	Funkcja	Domyślna wartość argumentu wykorzystana, jeśli wartość nie została przekazana w wywołaniu
def func(*nazwa)	Funkcja	Dopasowuje i zbiera pozostałe argumenty pozycyjne (w krotce)
def func(**nazwa)	Funkcja	Dopasowuje i zbiera pozostałe argumenty-słowa kluczowe (w słowniku)
def func(*argumenty, nazwa)	Funkcja	Argumenty, które w wywołaniu muszą być przekazane za pomocą słowa kluczowego (Python 3.x)
def func(*, nazwa=wartość)	Funkcja	Argumenty, które w wywołaniu muszą być przekazane za pomocą słowa kluczowego (Python 3.x)

Powysze specjalne tryby dopasowywania argumentów dzielą się na wywołania i definicje funkcji w następujący sposób:

- W wywołaniu funkcji (pierwszych czterech wierszach tabeli) proste wartości dopasowywane są po pozycji, jednak użycie formy *nazwa=wartość* mówi Pythonowi, że ma zamiast tego dopasować do argumentów po nazwie — są to zatem argumenty ze słowami kluczowymi. Użycie form **obiekt_iterowalny* lub ***słownik* w wywołaniu pozwala na spakowanie dowolnej liczby obiektów pozycyjnych lub słów kluczowych, odpowiednio, w sekwencjach (i innych obiektach iterowalnych) lub słownikach i rozpakowanie ich do postaci odrębnych, pojedynczych argumentów przy przekazaniu ich do funkcji.
- W nagłówku funkcji (reszta tabeli) prosta nazwa dopasowywana jest po pozycji lub nazwie, w zależności od sposobu przekazania przez kod wywołujący. Forma *nazwa=wartość* określa domyślną wartość argumentu. Forma **nazwa* zbiera dodatkowe niedopasowane argumenty pozycyjne w krotkę, a forma ***nazwa* zbiera dodatkowe argumenty-słowa kluczowe w słownik. W Pythonie 3.x wszystkie normalne nazwy argumentów lub argumenty z wartościami domyślnymi znajdujące się po **nazwa* lub samym znaku *** są argumentami mogącymi być tylko słowami kluczowymi i muszą w wywołaniach być przekazywane za pomocą słów kluczowych.

Z tych wszystkich propozycji w kodzie napisanym w Pythonie najczęściej używane są argumenty-słowa kluczowe oraz wartości domyślne. Z obu form korzystaliśmy już w sposób nieformalny we wcześniejszych częściach książki:

- Słowa kluczowe wykorzystaliśmy już do podawania opcji funkcji `print` z wersji 3.x, jednak mają one także bardziej uniwersalne zastosowanie. Słowa kluczowe pozwalają nam na opatrzenie dowolnego argumentu jego nazwą, tak by wywołania były bardziej znaczące.
- Z wartościami domyślnymi również spotkaliśmy się już wcześniej, jako ze sposobem przekazywania wartości z zasięgu funkcji zawierającej, jednak tak naprawdę także one są bardziej uniwersalne. Pozwalają nam na sprawienie, że każdy argument funkcji będzie opcjonalny, i podają wartość domyślną w definicji funkcji.

Jak zobaczymy wkrótce, połączenie wartości domyślnych w nagłówkach funkcji oraz słów kluczowych w wywołaniach pozwala nam wybierać, które wartości domyślne chcemy nadpisać.

Podsumowując, specjalne tryby dopasowania pozwalają na pewną swobodę w zakresie tego, ile argumentów musimy przekazać do funkcji. Jeśli funkcja określa wartości domyślne, będą one wykorzystane, kiedy przekażemy zbyt małą liczbę argumentów. Jeśli funkcja wykorzystuje formę zmiennej listy argumentów z ***, możemy przekazać zbyt dużą liczbę argumentów. Forma ta gromadzi dodatkowe argumenty w strukturach danych w celu przetworzenia ich w funkcji.

Dopasowywanie argumentów — szczegóły

Jeśli zdecydujemy się użyć specjalnych trybów dopasowywania argumentów i połączyc je ze sobą, Python wymaga przestrzegania następujących reguł dotyczących kolejności:

- W wywołaniu funkcji argumenty muszą pojawiać się w następującej kolejności: dowolne argumenty pozycyjne (*wartość*), po których następuje kombinacja argumentów ze słowami kluczowymi (*nazwa=wartość*), obiekt iterowalny, a na końcu ***słownik*.
- W nagłówku funkcji argumenty muszą się pojawiać w następującej kolejności — normalne argumenty (*wartość*), po nich argumenty ze słowami kluczowymi (*nazwa=wartość*), potem forma **nazwa* (lub *** w wersji 3.x), następnie argumenty mogące być tylko słowami kluczowymi *nazwa* lub *nazwa=wartość* (w wersji 3.x), a na końcu ***nazwa*.

Zarówno w wywołaniach, jak i nagłówkach forma ***args* musi się pojawiać na samym końcu. Jeżeli pomieszamy argumenty w innej kolejności, otrzymamy błąd składni, ponieważ inne kombinacje mogą być niejednoznaczne. Python wewnętrznie wykonuje następujące kroki mające na celu dopasowanie argumentów przed przypisaniem:

1. Przypisuje argumenty niebędące słowami kluczowymi zgodnie z pozycją.

2. Przypisuje argumenty będące słowami kluczowymi poprzez dopasowanie nazw.
3. Przypisuje dodatkowe argumenty niebędące słowami kluczowymi do krotki `*nazwa`.
4. Przypisuje dodatkowe argumenty będące słowami kluczowymi do słownika `**nazwa`.
5. Przypisuje do argumentów nieprzypisanych wartości domyślne z nagłówka.

Później Python sprawdza, czy do każdego argumentu przypisano dokładnie jedną wartość. Jeżeli tak nie jest, zgłaszany jest błąd. Po zakończeniu dopasowywania Python przypisuje nazwy argumentów do obiektów do nich przekazanych.

Sam algorytm dopasowywania wykorzystywany w Pythonie jest nieco bardziej skomplikowany (musi również wziąć pod uwagę na przykład argumenty będące słowami kluczowymi z wersji 3.x), dlatego po dokładny opis odsyłam do dokumentacji języka. Nie jest to lektura obowiązkowa, jednak prześledzenie mechanizmu dopasowania Pythona pozwoli nam lepiej zrozumieć bardziej zawiłe przypadki, w szczególności takie z mieszanymi trybami.

	<p>W Pythonie 3.x nazwy argumentów w nagłówku funkcji mogą także mieć wartości <i>adnotacji</i>, podane w formie <code>nazwa:wartość</code> (lub <code>nazwa:wartość=wartość_domyślna</code>, kiedy obecne są wartości domyślne). Jest to po prostu dodatkowa składnia przeznaczona dla argumentów, która nie rozszerza ani nie zmienia opisanych powyżej reguł kolejności argumentów. Sama funkcja także może mieć wartość adnotacji, podaną w postaci <code>f()>wartość</code>. Python dołącza wartości adnotacji do obiektu funkcji. Omówienie adnotacji funkcji możesz znaleźć w rozdziale 19.</p>
---	---

Przykłady ze słowami kluczowymi i wartościami domyślnymi

Wszystko to w kodzie jest znacznie prostsze, niż mógłby sugerować powyższy opis. Jeżeli nie korzystamy ze specjalnej składni dopasowania, Python dopasowuje nazwy po pozycji, od lewej do prawej strony, tak jak większość pozostałych języków programowania. Jeżeli na przykład zdefiniujemy funkcję wymagającą trzech argumentów, musimy wywołać ją z trzema argumentami.

```
>>> def f(a, b, c): print(a, b, c)
>>> f(1, 2, 3)
1 2 3
```

Tutaj przekazujemy je według pozycji — a dopasowywane jest do 1, b do 2, natomiast c do 3 (działa to tak samo w Pythonie 3.x i 2.x, jednak w wersji 2.x wyświetlane są dodatkowe nawiasy krotki, ponieważ korzystamy z wywołania `print` z wersji 3.x).

Słowa kluczowe

W Pythonie można jednak bardziej dokładnie określić, co ma gdzie trafić, kiedy wywołujemy funkcję. Argumenty ze słowami kluczowymi pozwalają na dopasowanie po *nazwie*, a nie pozycji. W przykładzie użyjemy tej samej funkcji:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

Element `c=3` w tym wywołaniu oznacza na przykład przesłanie wartości 3 do argumentu o nazwie `c`. Z bardziej formalnego punktu widzenia Python dopasowuje nazwę `c` w wywołaniu do

argumentu o nazwie `c` w nagłówku definicji funkcji, a następnie przekazuje wartość 3 do tego argumentu. Rezultat tego wywołania będzie taki jak poprzedniego, jednak warto zwrócić uwagę na to, że uporządkowanie argumentów od lewej do prawej strony w przypadku użycia słów kluczowych nie ma znaczenia — argumenty dopasowywane są po nazwie, a nie pozycji. Można nawet połączyć argumenty pozycyjne i słowa kluczowe w jednym wywołaniu. W takim przypadku wszystkie argumenty pozycyjne dopasowywane są od lewej do prawej strony nagłówka, zanim słowa kluczowe zostaną dopasowane po nazwach.

```
>>> f(1, c=3, b=2)          # a otrzymuje wartość 1 według pozycji, b i  
c przekazywane są po nazwie  
1 2 3
```

Kiedy większość użytkowników pierwszy raz widzi taki zapis, zastanawia się, do czego można go użyć. Słowa kluczowe pełnią w Pythonie dwie role. Po pierwsze sprawiają, że wywołania są nieco bardziej samodokumentujące (zakładając, że w kodzie funkcji użyliśmy bardziej znaczących nazw od `a`, `b` i `c`). Na przykład wywołanie w tej formie:

```
func(name='Teofil', age=40, job='programista')
```

jest bardziej znaczące od wywołania z trzema górnymi wartościami rozdzielonymi przecinkami, zwłaszcza w większych programach — słowa kluczowe służą jako podpisy danych z wywołania. Druga ważna dziedzina zastosowania słów kluczowych pojawia się w połączeniu z wartościami domyślnymi, do czego za chwilę dojdziemy.

Wartości domyślne

Mówiliśmy już nieco o wartościach domyślnych, kiedy omawialiśmy zasięg funkcji zagnieżdżonych. W skrócie, wartości domyślne pozwalają nam uczynić wybrane argumenty funkcji opcjonalnymi. Jeżeli do argumentu takiego nie przekaże się wartości, przed wykonaniem funkcji zostaje do niego przypisana wartość domyślna. Poniżej znajduje się przykład funkcji wymagającej jednego argumentu z dwoma wartościami domyślnymi.

```
>>> def f(a, b=2, c=3): print(a, b, c)      # a to argument wymagany, b i  
c są opcjonalne
```

Kiedy wywołujemy tę funkcję, musimy podać wartość zmiennej `a` albo po pozycji, albo za pomocą słowa kluczowego. Podanie wartości dla zmiennych `b` oraz `c` jest opcjonalne. Jeżeli nie przekażemy wartości do tych dwóch zmiennych, będą one miały wartości, odpowiednio, 2 i 3.

```
>>> f(1)                                     # Używamy wartości domyślnych  
1 2 3  
>>> f(a=1)  
1 2 3
```

Jeżeli przekażemy dwie wartości, jedynie `c` otrzymuje swoją wartość domyślną. W przypadku podania trzech wartości nie zostaną użyte żadne wartości domyślne.

```
>>> f(1, 4)                                 # Nadpisanie wartości  
domyślnych  
1 4 3  
>>> f(1, 4, 5)  
1 4 5
```

Wreszcie poniżej widać sposób interakcji pomiędzy słowami kluczowymi a wartościami domyślnymi. Ponieważ słowa kluczowe odwracają normalne odwzorowanie oparte na pozycji od

lewej do prawej strony, pozwalają one na przeskoczenie argumentów z wartościami domyślnymi.

```
>>> f(1, c=6) # Wybieranie wartości
domyślnych
1 2 6
```

W powyższym kodzie zmienna `a` otrzymuje wartość 1 zgodnie z pozycją, zmienna `c` otrzymuje wartość 6 ze względu na użycie słowa kluczowego, natomiast znajdująca się pomiędzy nimi zmienna `b` ma wartość domyślną równą 2.

Należy uważać, by nie pomylić specjalnej składni `nazwa=wartość` w nagłówku funkcji i wywołaniu funkcji. W wywołaniu oznacza ona argument słowa kluczowego dopasowanego po nazwie, natomiast w nagłówku określa wartość domyślną opcjonalnego argumentu. W obu przypadkach nie jest to instrukcja przypisania (pomimo że tak wygląda), a jedynie specjalna składnia dla tych dwóch kontekstów, modyfikująca domyślny mechanizm dopasowywania argumentów.

Łączenie słów kluczowych i wartości domyślnych

Poniżej znajduje się nieco większy przykład demonstrujący działanie słów kluczowych oraz wartości domyślnych. W poniższym kodzie obiekt wywołujący musi zawsze przekazać co najmniej dwa argumenty (by pasowały one do zmiennych `spam` i `eggs`), jednak dwa pozostałe są opcjonalne. Jeżeli zostają pominięte, Python przypisuje do argumentów `toast` i `ham` wartości domyślne podane w nagłówku.

```
def func(spam, eggs, toast=0, ham=0): # Pierwsze dwa wymagane
    print((spam, eggs, toast, ham))
func(1, 2) # Wynik: (1, 2, 0, 0)
func(1, ham=1, eggs=0) # Wynik: (1, 0, 0, 1)
func(spam=1, eggs=0) # Wynik: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3) # Wynik: (3, 2, 1, 0)
func(1, 2, 3, 4) # Wynik: (1, 2, 3, 4)
```

Warto ponownie zwrócić uwagę na to, że kiedy w wywołaniu wykorzystane są argumenty ze słowami kluczowymi, kolejność wymienionych argumentów nie ma znaczenia. Python dopasowuje je po nazwie, a nie pozycji. Kod wywołujący musi podać wartości dla argumentów `spam` oraz `eggs`, jednak mogą one zostać dopasowane albo po pozycji, albo po nazwie. Warto również pamiętać, że forma `nazwa=wartość` oznacza co innego w wywołaniu i w instrukcji `def` (w wywołaniu słowo kluczowe, a w nagłówku funkcji — wartość domyślną).



Uwaga na mutowalne wartości domyślne: zgodnie z przypisem w poprzednim rozdziale, jeżeli w kodzie używasz wartości domyślnej będącej obiektem mutowalnym (np. `def f(a = [])`), ten sam pojedynczy obiekt mutowalny jest ponownie wykorzystywany za każdym razem, gdy funkcja jest później wywoływana — nawet jeżeli zostanie zmieniony w miejscu w funkcji. Efektem netto jest to, że domyślna wartość argumentu zachowuje wartość z poprzedniego wywołania i nie jest resetowana do pierwotnej wartości zakodowanej w nagłówku `def`. Aby zresetować go na nowo przy każdym wywołaniu, powinieneś przenieść przypisanie do ciała funkcji. Mutowalne wartości domyślne pozwalają na zachowanie stanu, ale często sprawiają niespodzianki. Ponieważ jest to tak powszechnie spotykana pułapka, odłożymy dalsze omawianie tego zagadnienia aż do listy potencjalnych „potrzasków” na końcu rozdziału 21.

Przykłady dowolnych argumentów

Ostatnie dwa rozszerzenia dopasowania, * oraz **, zaprojektowane zostały z myślą o obsłudze funkcji, które przyjmują *dowolną* liczbę argumentów. Oba mogą się pojawiać albo w definicji funkcji, albo w jej wywołaniu i mają w tych lokalizacjach powiązane cele.

Nagłówki: zbieranie argumentów

Pierwsze zastosowanie — w definicji funkcji — zbiera niedopasowane argumenty *pozycyjne* w krotkę.

```
>>> def f(*args): print(args)
```

Kiedy funkcja zostaje wywołana, Python zbiera wszystkie argumenty w nową krotkę i przypisuje zmienną args do tej krotki. Ponieważ jest to normalny obiekt krotki, można go indeksować czy na przykład przechodzić za pomocą pętli for.

```
>>> f()  
()  
>>> f(1)  
(1,)  
>>> f(1,2,3,4)  
(1, 2, 3, 4)
```

Opcja ** jest podobna, jednak działa tylko dla argumentów ze *słowami kluczowymi* — zbiera je w nowy słownik, który może następnie zostać przetworzony za pomocą normalnych narzędzi słowników. W pewnym sensie forma z ** pozwala na konwersję ze słów kluczowych na słowniki, które można następnie przechodzić na przykład za pomocą wywołań keys, iteratorów słowników itp. (to mniej więcej to, co robi wywołanie funkcji dict po przekazaniu słów kluczowych, ale forma ** zwraca nowy słownik):

```
>>> def f(**args): print(args)  
  
>>> f()  
{}  
>>> f(a=1, b=2)  
{'a': 1, 'b': 2}
```

Wreszcie nagłówki funkcji mogą łączyć normalne argumenty * oraz ** w celu zaimplementowania bardzo elastycznych sygnatur wywołań. Przykładowo w poniższym kodzie 1 przekazywane jest do a według pozycji, 2 i 3 zbierane są w krotkę pozycyjną pargs, natomiast x i y trafiają do słownikaków słów kluczowych kargs:

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)  
  
>>> f(1, 2, 3, x=1, y=2)  
1 (2, 3) {'y': 2, 'x': 1}
```

Takie rozwiązanie jest jednak spotykane niezbyt często, niemniej jednak pojawia się w funkcjach, które muszą obsługiwać wiele wzorców wywołań (na przykład dla zachowania kompatybilności wstępnej). Tak naprawdę opcje te można łączyć na jeszcze bardziej

skomplikowane sposoby, które mogą się na pierwszy rzut oka wydawać niejednoznaczne — zajmiemy się tym w dalszej części niniejszego rozdziału. Najpierw jednak zobaczymy, co stanie się, kiedy * oraz ** umieszczone są w wywołaniach funkcji, a nie ich definicjach.

Wywołania: rozpakowywanie argumentów

We wszystkich nowszych wersjach Pythona składni * można również użyć przy wywoływaniu funkcji. W tym kontekście jej znaczenie jest odwrotnością znaczenia z definicji funkcji — rozpakowuje kolekcję argumentów, zamiast ją budować. Możemy na przykład przekazać do funkcji cztery argumenty w krotce i pozwolić Pythonowi na rozpakowanie krotki na pojedyncze argumenty.

```
>>> def func(a, b, c, d): print(a, b, c, d)
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)                                     # To samo co wywołanie func(1,
2, 3, 4)
1 2 3 4
```

W podobny sposób składnia ** w wywołaniu funkcji rozpakowuje słownik par klucz-wartość na pojedyncze argumenty ze słowami kluczowymi.

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)                                     # To samo co wywołanie func(a=1,
b=2, c=3, d=4)
1 2 3 4
```

I znów w wywołaniu możemy na wiele elastycznych sposobów połączyć normalne argumenty pozycyjne oraz argumenty ze słowami kluczowymi.

```
>>> func(*(1, 2), **{'d': 4, 'c': 4})           # To samo co wywołanie func(1,
2, d=4, c=3)
1 2 4 4
>>> func(1, *(2, 3), **{'d': 4})                 # To samo co wywołanie func(1,
2, 3, d=4)
1 2 3 4
>>> func(1, c=3, *(2,), **{'d': 4})             # To samo co wywołanie func(1,
2, c=3, d=4)
1 2 3 4
>>> func(1, *(2, 3), d=4)                         # To samo co wywołanie func(1,
2, 3, d=4)
1 2 3 4
>>> f(1, *(2,), c=3, **{'d': 4})                 # To samo co wywołanie func(1,
2, c=3, d=4)
1 2 3 4
```

Ten rodzaj kodu jest wygodny, kiedy nie możemy w momencie pisania skryptu przewidzieć liczby argumentów, które będą przekazane do funkcji. Możemy zamiast tego zbudować kolekcję

argumentów w czasie wykonania i w ten sposób wywołać funkcję w sposób uniwersalny. Ponownie nie należy mylić składni z * oraz ** w nagłówku funkcji i jej wywołaniu. W nagłówku służy do zebrania dowolnej liczby argumentów, natomiast w wywołaniu rozpakowuje dowolną liczbę argumentów. W obu przypadkach * oznacza argumenty pozycyjne, a ** odnosi się do argumentów ze słowami kluczowymi.



Jak widzieliśmy w rozdziale 14., forma *pargs w wywołaniu jest *kontekstem iteracyjnym*, dlatego przyjmuje ona dowolny obiekt, na którym można wykonywać iterację, a nie tylko krotki i inne sekwencje, zgodnie z przykładami zaprezentowanymi wyżej. Po znaku * można na przykład zastosować obiekt pliku, co powoduje rozpakowanie jego wierszy do pojedynczych argumentów (na przykład func(*open('nazwa_pliku'))). Dodatkowe przykłady zastosowania takich argumentów przedstawimy w rozdziale 20., po omówieniu generatorów.

Taka uniwersalność obsługiwana jest zarówno w Pythonie 3.x, jak i 2.x, jednak jest prawdziwa tylko w przypadku wywołań. Forma *pargs w wywołaniu pozwala na zastosowanie dowolnego obiektu, na którym można wykonywać iterację, jednak ta sama forma w nagłówku instrukcji def zawsze zbiera dodatkowe argumenty w krotkę. Takie zachowanie nagłówka jest podobne w duchu i składni do * w rozszerzonych formach przypisów rozpakowujących sekwencje z Pythona 3.x, omówionych w rozdziale 11. (na przykład x, *y = z), choć opcja ta zawsze tworzy listy, a nie krotki.

Ogólne zastosowanie funkcji

Przykłady z poprzedniego podrozdziału mogą się wydawać nieco akademickie (jeżeli nie wręcz ezoteryczne), jednak wykorzystywane są częściej, niż można by się tego spodziewać. Niektóre programy muszą wywoływać różne funkcje w sposób uniwersalny, nie znając ich nazw czy argumentów z wyprzedzeniem. Tak naprawdę prawdziwa siła specjalnej składni wywołania z dowolnymi argumentami tkwi w braku konieczności posiadania przed napisaniem skryptu wiedzy, ile argumentów wymaga wywołanie funkcji. Możemy na przykład wykorzystać polecenie if do wybierania elementów ze zbioru funkcji i list argumentów, a następnie wywołania ich w ogólny sposób (funkcje w niektórych przykładach poniżej są dosyć hipotetyczne):

```
if dowolnytest:  
    action, args = func1, (1,)                      # W tym przypadku wywołanie func1  
    z 1 argumentem  
  
else:  
    action, args = func2, (1, 2, 3)                  # Tutaj wywołanie func2 z 3  
    argumentami  
  
...  
  
action(*args)                                     # Uniwersalne wywołanie
```

Powyższy przykład wykorzystuje zarówno formę *, jak i fakt, że funkcje są obiektami, do których można odwoływać się i wywoływać dowolną zmienną. Mówiąc bardziej ogólnie, składnia wywołań z dowolną liczbą argumentów przydatna jest zawsze wtedy, gdy nie możemy przewidzieć listy argumentów. Jeżeli użytkownik wybierze na przykład dowolną funkcję za pośrednictwem interfejsu użytkownika, niemożliwe może się okazać napisanie wywołania funkcji w czasie tworzenia skryptu. Aby obejść to ograniczenie, wystarczy zbudować listę argumentów za pomocą działań na sekwencjach i wywołać ją z nazwami ze znakami * w celu rozpakowania argumentów.

```
>>> ...zdefiniuj lub zimportuj funkcję func3...  
>>> args = (2,3)
```

```
>>> args += (4,)  
>>> args  
(2, 3, 4)  
>>> func3(*args)
```

Ponieważ lista argumentów przekazywana jest tutaj jako krotka, program może budować ją w czasie wykonania. Technika ta przydaje się także w przypadku funkcji, które sprawdzają lub mierzą czas innych funkcji. Przykładowo w poniższym kodzie obsługujemy dowolną funkcję z dowolną liczbą argumentów, przekazując dowolne argumenty, jakie zostały przesłane (kod źródłowy tej funkcji znajdziesz w pliku o nazwie *tracer0.py* w pakietie przykładów):

```
def tracer(func, *pargs, **kargs):          # Przyjmuje dowolne argumenty  
    print('wywoływanie:', func.__name__)  
    return func(*pargs, **kargs)                # Przekazuje dowolne argumenty  
  
def func(a, b, c, d):  
    return a + b + c + d  
  
print(tracer(func, 1, 2, c=3, d=4))
```

Przedstawiony przykład korzysta z wbudowanego atrybutu `__name__` dołączonego do każdej funkcji (jak można się spodziewać, jest to ciąg reprezentujący nazwę funkcji) i używa gwiazdki do gromadzenia, a następnie rozpakowywania argumentów przeznaczonych dla śledzonej funkcji. Innymi słowy, po uruchomieniu tego kodu argumenty są przechwytywane przez moduł śledzący, a następnie *propagowane* za pomocą składni wywołania z dowolną liczbą argumentów:

```
wywoływanie: func  
10
```

Inny przykład tej techniki możesz znaleźć na końcu poprzedniego rozdziału, w którym wykorzystano ją do zresetowania wbudowanej funkcji `open`. Dodatkowe przykłady takich zastosowań pokażemy w dalszej części tej książki; zobacz zwłaszcza przykłady synchronizacji sekwencji w rozdziale 21. i różne narzędzia związane z dekoratorami, które omówimy w rozdziale 39. Jest to popularna technika stosowana w narzędziach ogólnego przeznaczenia.

Zlikwidowana wbudowana funkcja `apply` (Python 2.x)

Przed wersją 3.x Pythona efekt działania składni z dowolną liczbą argumentów `*args` oraz `**args` można było uzyskać za pomocą wbudowanej funkcji o nazwie `apply`. Technika ta została usunięta z wersji 3.x, ponieważ jest teraz zbędna (w wersji tej usunięto wiele zakurzonych narzędzi, które nagromadziły się przez lata). Jest ona jednak nadal dostępna w Pythonie 2.x i można na nią natrafić w starszym kodzie napisanym w wersjach 2.x.

Mówiąc w skrócie, poniższe wiersze są równoważne w Pythonie przed wersją 3.x:

```
func(*pargs, **kargs)          # Nowsza składnia wywołania:  
func(*sekwencja, **słownik)  
  
apply(func, pargs, kargs)      # Zlikwidowana funkcja wbudowana:  
apply(func, sekwencja, słownik)
```

Przykładowo zastanówmy się nad poniższą funkcją, która przyjmuje dowolną liczbę argumentów pozycyjnych lub będących słowami kluczowymi:

```
>>> def echo(*args, **kwargs): print(args, kwargs)
```

```
>>> echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

W Pythonie 2.x możemy wywołać ją w sposób uniwersalny za pomocą `apply` lub z użyciem składni wywołania wymaganej obecnie w wersji 3.x:

```
>>> pargs = (1, 2)
>>> kargs = {'a':3, 'b':4}
>>> apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}
>>> echo(*pargs, **kargs)
(1, 2) {'a': 3, 'b': 4}
```

Obie formy działają również w przypadku funkcji wbudowanych w wersji 2.x (zauważ, że wersja dodaje znak L na końcu długich liczb całkowitych):

```
>>> apply(pow, (2, 100))
1267650600228229401496703205376L
>>> pow(*(2, 100))
1267650600228229401496703205376L
```

Składnia wywołania z rozpakowaniem jest nowsza od funkcji `apply`, ogólnie zalecana i wymagana w wersji 3.x (technicznie rzecz biorąc, składnia ta została dodana w wersji 2.0; funkcja została uznana za przestarzałą w 2.3, nadal można jej używać bez ostrzeżeń w wersji 2.7, ale zniknęła na dobre w wersji 3.x.). Poza podobieństwem do kolektorów `*pargs` i `**kargs` z nagłówków instrukcji `def` oraz faktem, że wymaga mniej pisania, nowsza składnia wywołania pozwala także na przekazywanie dodatkowych argumentów bez konieczności ręcznego rozszerzania sekwencji bądź słowników argumentów.

```
>>> echo(0, c=5, *pargs, **kargs)           # Normalny, słowo kluczowe,
*sekwencja, **słownik
(0, 1, 2) {'a': 3, 'c': 5, 'b': 4}
```

Oznacza to, że składnia wywołania jest *bardziej ogólna*. Ponieważ jest też wymagana w wersji 3.x, teraz powinieneś wyrzucić z głowy całą swoją wiedzę na temat funkcji `apply` (o ile oczywiście nie pojawia się ona w kodzie Pythona 2.x, z którego musisz korzystać lub który musisz utrzymywać...).

Argumenty tylko ze słowami kluczowymi (z Pythona 3.x)

W Pythonie 3.0 uogólniono reguły kolejności w nagłówkach funkcji w taki sposób, by możliwe było podawanie argumentów *mogących być tylko słowami kluczowymi* (ang. *keyword-only argument*) — czyli argumentów, które trzeba przekazywać za pomocą słowa kluczowego i które nigdy nie zostaną wypełnione przez jakikolwiek argument pozycyjny. Jest to przydatne, jeśli chcemy, by funkcja zarówno przetwarzała dowolną liczbę argumentów, jak i przyjmowała możliwie opcjonalne opcje konfiguracyjne.

Z punktu widzenia składni argumenty tego typu zapisywane są w kodzie jako nazwane argumenty, które na liście argumentów mogą pojawić się po formie `*args`. Wszystkie muszą być przekazywane za pomocą składni ze słowami kluczowymi w wywołaniu. Przykładowo w

poniższym kodzie a można przekazać za pomocą nazwy lub pozycji, b zbiera wszystkie dodatkowe argumenty pozycyjne, natomiast c można przekazać jedynie przez słowo kluczowe. W wersji 3.x wygląda to tak:

```
>>> def kwonly(a, *b, c):  
    print(a, b, c)  
>>> kwonly(1, 2, c=3)  
1 (2,) 3  
>>> kwonly(a=1, c=3)  
1 () 3  
>>> kwonly(1, 2, 3)  
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
```

Możemy również wykorzystać sam znak * w liście argumentów w celu wskazania, że funkcja nie przyjmuje listy argumentów o zmiennej długości, natomiast nadal oczekuje, że wszystkie argumenty pojawiające się po * zostaną przekazane jako słowa kluczowe. W poniższej funkcji a można znowu przekazać za pomocą nazwy lub pozycji, natomiast b oraz c muszą być słowami kluczowymi i nie są dozwolone żadne dodatkowe argumenty pozycyjne:

```
>>> def kwonly(a, *, b, c):  
    print(a, b, c)  
>>> kwonly(1, c=3, b=2)  
1 2 3  
>>> kwonly(c=3, b=2, a=1)  
1 2 3  
>>> kwonly(1, 2, 3)  
TypeError: kwonly() takes 1 positional argument but 3 were given  
>>> kwonly(1)  
TypeError: kwonly() missing 2 required keyword-only arguments: 'b' and 'c'
```

W przypadku argumentów mogących być tylko słowami kluczowymi nadal możemy wykorzystać wartości domyślne, choć pojawiają się one po znaku * w nagłówku funkcji. W poniższym kodzie a można przekazać za pomocą nazwy lub pozycji, natomiast b oraz c są opcjonalne, jednak jeśli zostaną użyte, muszą być przekazane za pomocą słowa kluczowego:

```
>>> def kwonly(a, *, b='mielonka', c='szynka'):  
    print(a, b, c)  
>>> kwonly(1)  
1 mielonka szynka  
>>> kwonly(1, c=3)  
1 mielonka 3  
>>> kwonly(a=1)  
1 mielonka szynka
```

```

>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given

Tak naprawdę argumenty mogące być tylko słowami kluczowymi z wartościami domyślnymi są opcjonalne, jednak te bez wartości domyślnych stają się wymaganymi słowami kluczowymi dla funkcji:

>>> def kwonly(a, *, b, c='mielonka'):
    print(a, b, c)

>>> kwonly(1, b='jajka')
1 jajka mielonka
>>> kwonly(1, c='jajka')
TypeError: kwonly() missing 1 required keyword-only argument: 'b'
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
>>> def kwonly(a, *, b=1, c, d=2):
    print(a, b, c, d)

>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given

```

Reguły dotyczące kolejności

Wreszcie warto podkreślić, że argumenty mogące być tylko słowami kluczowymi muszą być podawane po pojedynczym znaku *, a nie dwóch — po formie z dowolnymi słowami kluczowymi **args nie mogą się pojawiać nazwane argumenty. Znaki ** nie mogą też pojawić się samodzielnie w liście argumentów. Oba zapisy generują błąd składni:

```

>>> def kwonly(a, **pargs, b, c):
SyntaxError: invalid syntax
>>> def kwonly(a, **, b, c):
SyntaxError: invalid syntax

```

Oznacza to, że w *nagłówku* funkcji argumenty mogące być tylko słowami kluczowymi muszą być zapisywane przed formą z dowolnymi słowami kluczowymi `**args`, a po formie `*args` z dowolnymi argumentami pozycyjnymi — o ile obie są obecne. Za każdym razem, gdy przed `*args` pojawia się nazwa argumentu, będzie to raczej domyślny argument pozycyjny, a nie argument mogący być tylko słowem kluczowym:

```
>>> def f(a, *b, **d, c=6): print(a, b, c, d) # Przed ** tylko argumenty
      będące słowami kluczowymi!
SyntaxError: invalid syntax

>>> def f(a, *b, c=6, **d): print(a, b, c, d) # Zebranie argumentów w
      nagłówku

>>> f(1, 2, 3, x=4, y=5)                                # Użyte wartości domyślne
1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, 2, 3, x=4, y=5, c=7)                          # Nadpisanie wartości
      domyślnych
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, 2, 3, c=7, x=4, y=5)                          # W dowolnym miejscu w słowach
      kluczowych
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> def f(a, c=6, *b, **d): print(a, b, c, d) # c nie jest argumentem
      będącym słowem kluczowym!

...
>>> f(1, 2, 3, x=4)
1 (3,) 2 {'x': 4}
```

Tak naprawdę podobne reguły kolejności są prawdziwe dla *wywołań* funkcji. Kiedy przekazywane są argumenty mogące być tylko słowami kluczowymi, muszą się one pojawiać przed formą `**args`. Argument mogący być tylko słowem kluczowym może jednak być zapisany albo przed formą `*args`, albo po niej, i może się mieścić w formie `**args`:

```
>>> def f(a, *b, c=6, **d): print(a, b, c, d) # Pomiędzy * a ** tylko słowa
      kluczowe

...
>>> f(1, *(2, 3), **dict(x=4, y=5))          # Rozpakowanie argumentów przy
      wywołaniu
1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, *(2, 3), **dict(x=4, y=5), c=7)    # Słowa kluczowe przed
      **argumenty!

SyntaxError: invalid syntax

>>> f(1, *(2, 3), c=7, **dict(x=4, y=5))    # Nadpisanie wartości
      domyślnych
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, c=7, *(2, 3), **dict(x=4, y=5))    # Przed lub po *
```

```
1 (2, 3) 7 {'y': 5, 'x': 4}
>>> f(1, *(2, 3), **dict(x=4, y=5, c=7))      # Argument mogący być tylko
słownem kluczowym w **
1 (2, 3) 7 {'y': 5, 'x': 4}
```

Spróbuj przeanalizować te przypadki samodzielnie w połączeniu z ogólnymi zasadami porządkowania argumentów opisanymi wcześniej. Na pierwszy rzut oka może się wydawać, że są to nieco dziwaczne przypadki oparte na sztucznych przykładach, ale w rzeczywistości mogą być przydatne w praktyce, szczególnie dla osób piszących biblioteki i narzędzi, z których mogą korzystać inni programiści Pythona.

Czemu służą argumenty ze słowami kluczowymi?

Po co nam więc argumenty ze słowami kluczowymi? W skrócie, umożliwiają one funkcji przyjmowanie zarówno dowolnej liczby argumentów pozycyjnych do przetworzenia, jak i przekazywanie opcji konfiguracyjnych w postaci słów kluczowych. Choć ich stosowanie jest opcjonalne, bez argumentów ze słowami kluczowymi podanie wartości domyślnych dla takich opcji i upewnienie się, że nie przekazano żadnych nadmiarowych słów kluczowych, może wymagać dodatkowej pracy.

Wyobraźmy sobie funkcję przetwarzającą zbiór przekazanych obiektów i pozwalającą na przekazanie opcji śledzącej:

```
process(X, Y, Z)                      # Wykorzystanie wartości
domyślnej opcji

process(X, Y, notify=True)             # Nadpisanie wartości
domyślnej opcji
```

Bez argumentów ze słowami kluczowymi musimy użyć zarówno formy `*args`, jak i `**args` i ręcznie badać słowa kluczowe, natomiast po wykorzystaniu rozwiązania z argumentami ze słowami kluczowymi wymagana jest mniejsza ilość kodu. Poniższy kod gwarantuje, że żadne argumenty pozycyjne nie zostaną błędnie dopasowane do `notify`, i wymaga, by przy przekazaniu było to słowo kluczowe:

```
def process(*args, notify=False): ...
```

Ponieważ bardziej realistyczne przykłady tej techniki zobaczymy w dalszej części rozdziału, w sekcji „Emulacja funkcji print z Pythona 3.x”, resztę informacji zamieścimy właśnie tam. Dodatkowy przykład działania argumentów ze słowami kluczowymi znajduje się w studium przypadku z poniarem opcji iteracji w rozdziale 21. Dodatkowe ulepszenia definicji funkcji w Pythonie 3.x znajdują się przy omówieniu składni adnotacji funkcji w rozdziale 19.

Przykład z funkcją obliczającą minimum

No dobrze, najwyższy czas na coś bardziej realistycznego. Aby jakoś skonkretyzować koncepcje przedstawione w niniejszym rozdziale, wykonajmy ćwiczenie demonstrujące praktyczne zastosowanie narzędzi dopasowujących argumenty.

Założymy, że chcemy utworzyć kod funkcji potrafiącej obliczyć wartość minimalną dla dowolnego zbioru argumentów i dowolnego zbioru typów danych obiektów. Funkcja ta powinna zatem przyjmować zero lub większą liczbę argumentów — tyle, ile będziemy jej chcieli przekazać. Co więcej, funkcja ta powinna działać na wszystkich rodzajach typów obiektów Pythona: liczbach, łańcuchach znaków, listach, listach słowników, plikach, a nawet `None` (szczerze mówiąc, użytkownicy Pythona 3.x nie muszą obsługiwać słowników, ponieważ ich słowniki nie obsługują bezpośrednich porównań; patrz rozdziały 8. i 9.).

Pierwsze wymaganie staje się naturalnym polem do zastosowania opcji z * — możemy zebrać argumenty w krotkę i przejść każdy z nich po kolejno za pomocą prostej pętli `for`. Druga część definicji problemu jest prosta: ponieważ każdy typ obiektu obsługuje porównania, nie musimy dostosowywać funkcji do specyficznych wymagań poszczególnych typów (zastosowanie *polimorfizmu*) — możemy po prostu ślepo zestawić obiekty i pozwolić Pythonowi na wykonanie odpowiedniego dla nich porównania.

Pełne rozwiązanie

Poniższy plik pokazuje trzy sposoby kodowania tej operacji, z których co najmniej jeden został zasugerowany przez jednego z moich studentów (obecnie często wykorzystujemy ten przykład jako ćwiczenie grupowe mające na celu wyrwanie moich studentów z drzemki po obiedzie):

- Pierwsza funkcja pobiera pierwszy argument (`args` jest krotką) i przechodzi resztę, odcinając pierwszy argument (nie ma sensu porównywać obiektu z samym sobą, w szczególności jeśli może on być większą strukturą).
- Druga wersja pozwala Pythonowi na automatyczne wybranie pierwszego i pozostałych argumentów, zatem unika indeksowania i wycinka.
- Trzecia metoda dokonuje konwersji krotki na listę za pomocą wywołania wbudowanej funkcji `list` i stosuje metodę listy `sort`.

Metoda `sort` napisana jest jako kod języka C, dlatego czasami może być szybsza od pozostałych rozwiązań, jednak w większości przypadków liniowe przeglądanie z dwóch pierwszych technik będzie działać szybciej [1]. Plik `mins.py` zawiera kod wszystkich trzech rozwiązań.

```
def min1(*args):  
    res = args[0]  
    for arg in args[1:]:  
        if arg < res:  
            res = arg  
    return res  
  
def min2(first, *rest):  
    for arg in rest:  
        if arg < first:  
            first = arg  
    return first  
  
def min3(*args):  
    tmp = list(args) # Lub w Pythonie 2.4+: return  
    sorted(args)[0]  
    tmp.sort()  
    return tmp[0]  
  
print(min1(3,4,1,2))  
print(min2("bb", "aa"))  
print(min3([2,2], [1,1], [3,3]))
```

Wszystkie trzy rozwiązania dają po wykonaniu pliku ten sam wynik. Można spróbować wpisać kilka wywołań w sesji interaktywnej w celu samodzielnego eksperymentowania z trzema funkcjami.

```
% python mins.py
```

```
1
```

```
aa
```

```
[1, 1]
```

Warto zauważyć, że żaden z trzech wariantów nie sprawdza przypadku, w którym nie przekazano żadnych argumentów. Mogłyby to robić, ale nie ma to tutaj sensu — we wszystkich trzech rozwiązańach Python automatycznie zgłasza wyjątek, jeżeli do funkcji nie przekazano żadnych argumentów. Pierwszy wariant zwraca wyjątek, kiedy próbujemy pobrać element zerowy, drugi — kiedy Python wykrywa niedopasowanie listy argumentów, natomiast trzeci — kiedy próbujemy na końcu zwrócić element zerowy.

Właśnie to chcemy osiągnąć — ponieważ funkcje obsługują dowolny typ danych, nie istnieje poprawna wartość ostrzegawcza, którą moglibyśmy przekazać w celu oznaczenia błędu, więc równie dobrze możemy pozwolić na zgłoszenie wyjątku. Istnieją wyjątki od tej reguły (np. możesz samodzielnie sprawdzać argumenty pod kątem błędów, jeżeli wolisz wykonywać odpowiednią akcję przed dotarciem do kodu, który automatycznie zgłasza wyjątek), ale ogólnie lepiej jest założyć, że argumenty będą działać w kodzie funkcji, i pozwolić Pythonowi na zgłoszanie błędów, kiedy tak nie będzie.

Dodatkowy bonus

Punkty dodatkowe można uzyskać za modyfikację funkcji w taki sposób, by obliczały one wartości *maksymalne*, a nie minimalne. To proste zadanie — pierwsze dwie wersje wymagają jedynie zmiany < na >, natomiast trzecia wymaga zwrócenia `tmp[-1]` w miejsce `tmp[0]`. Aby uzyskać dodatkowy punkt, należy również pamiętać o zmianie nazwy funkcji na `max` (choć jest to całkowicie opcjonalne).

Można również utworzyć jedną uogólnioną funkcję obliczającą wartość minimalną lub maksymalną poprzez obliczenie wyrażenia porównania za pomocą narzędzi takich jak wbudowana funkcja `eval` (więcej informacji znajdziesz w dokumentacji biblioteki standardowej oraz różnych sekcjach tej książki, zwłaszcza w rozdziale 10.) lub przekazanie dowolnej funkcji porównujcej. Plik `minmax.py` pokazuje, w jaki sposób można zaimplementować ten ostatni mechanizm.

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y                      # Zobacz również lambda, eval
def grtrthan(x, y): return x > y
print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Kod samotestu
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

```
% python minmax.py  
1  
6
```

Funkcje są kolejnym typem obiektu, jaki może zostać przekazany do funkcji, podobnie jak pokazano w kodzie powyżej. Aby zrobić z tego funkcję `max` (czy dowolną inną), moglibyśmy na przykład przekazać właściwy rodzaj funkcji `test`. Może wymagać nieco dodatkowej pracy, jednak najważniejszą zaletą uogólniania funkcji (w miejsce kopowania i wklejania w celu zmiany jednego znaku) jest to, że w przyszłości będziemy musieli modyfikować tylko jeden kod, a nie dwa.

Puenta

Wszystko to było oczywiście jedynie ćwiczeniem programistycznym. Tak naprawdę nie ma powodu tworzyć w Pythonie funkcji `min` czy `max`, ponieważ obie są wbudowane w sam język! Spotkaliśmy je w rozdziale 5. w połączeniu z narzędziami działającymi na liczbach, a także w rozdziale 14. przy omawianiu kontekstów iteracyjnych. Wersje wbudowane działają prawie dokładnie tak samo jak nasze, jednak w celu uzyskania optymalnej szybkości napisane zostały w języku C i przyjmują albo jeden, albo większą liczbę argumentów, na których można wykonywać iterację. Nawet jeżeli w tym kontekście kod ten był nieco zbędny, sam wykorzystany tutaj wzorzec programowania może się przydać w innych scenariuszach.

Uogólnione funkcje działające na zbiorach

Przyjrzyjmy się teraz bardziej użytkowemu przykładowi działania specjalnych trybów dopasowywania argumentów. Na końcu rozdziału 16. utworzyliśmy funkcję zwracającą część wspólną dwóch sekwencji (wybierała ona elementy pojawiające się w obu sekwencjach). Poniżej znajduje się jej wersja obliczająca część wspólną dla dowolnej liczby sekwencji (jednej lub większej ich liczby), wykorzystująca postać dopasowania ze zmienną liczbą argumentów (`*args`) do zebrania wszystkich przekazanych argumentów. Ponieważ argumenty trafiają do funkcji w postaci krotki, możemy je przetworzyć za pomocą prostej pętli `for`. Dla zabawy napiszemy funkcję obliczającą sumę zbiorów i również przyjmującą dowolną liczbę argumentów oraz zbierającą elementy pojawiające się w dowolnym z argumentów.

```
def intersect(*args):  
    res = []  
    for x in args[0]:                      # Przejrzenie pierwszej  
        sekwencji  
        if x in res: continue                # Pomijamy duplikaty  
        for other in args[1:]:                  # Dla wszystkich pozostałych  
            argumentów  
            if x not in other: break          # Element w każdym z nich?  
            else:                            # Nie – wyjście z pętli  
                res.append(x)                  # Tak – dodanie elementów na  
                końca
```

```

    return res

def union(*args):
    res = []
    for seq in args:                                # Dla wszystkich argumentów
        for x in seq:                               # Dla wszystkich węzłów
            if not x in res:
                res.append(x)                      # Dodanie nowych elementów do
                                                wyniku
    return res

```

Ponieważ narzędzia te potencjalnie mogą nam się przydać w przyszłości (a na dodatek są zbyt duże, by wpisywać je ponownie w sesji interaktywnej), zachowamy te funkcje w pliku modułu o nazwie *inter2.py* (osoby, które nie pamiętają, jak działa importowanie modułów, mogą wrócić do wprowadzenia w rozdziale 3. lub poczekać na więcej informacji na temat modułów w piątej części książki). W obu funkcjach argumenty przekazane w wywołaniu przychodzą jako krotka `args`. Jak oryginalna funkcja `intersect`, obie funkcje działają na dowolnym typie sekwencji. Poniżej widać, jak funkcje te przetwarzająłańcuchy znaków, typy mieszane oraz liczbę sekwencji większą od dwóch.

```

% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "Teodor", "Teofil", "Troll"
>>> intersect(s1, s2), union(s1, s2)          # Dwa argumenty
(['T', 'e', 'o', 'o'], ['T', 'e', 'o', 'd', 'r', 'f', 'i', 'l'])
>>> intersect([1,2,3], (1,4))                 # Typy mieszane
[1]
>>> intersect(s1, s2, s3)                     # Trzy argumenty
['T', 'o', 'o']
>>> union(s1, s2, s3)
['T', 'e', 'o', 'd', 'r', 'f', 'i', 'l']

```

Aby dokładniej przetestować omawiane techniki, w kodzie przedstawionym poniżej tworzymy funkcję, która używa dwóch narzędzi do przetwarzania argumentów w różnej kolejności za pomocą prostej techniki mieszania, pokazanej w rozdziale 13. — funkcja ta korzysta z wycinków do przesuwania pierwszego elementu na koniec w każdej iteracji pętli, a następnie używa * do rozpakowania argumentów i sortuje wyniki tak, aby można je było porównać:

```

>>> def tester(func, items, trace=True):
    for i in range(len(items)):
        items = items[1:] + items[:1]
        if trace: print(items)
        print(sorted(func(*items)))
>>> tester(intersect, ('a', 'abcdefg', 'abdst', 'albmcnd'))

```

```

('abcdefg', 'abdst', 'albmcnd', 'a')
['a']
('abdst', 'albmcnd', 'a', 'abcdefg')
['a']
('albmcnd', 'a', 'abcdefg', 'abdst')
['a']
('a', 'abcdefg', 'abdst', 'albmcnd')
['a']

>>> tester(union, ('a', 'abcdefg', 'abdst', 'albmcnd'), False)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
>>> tester(intersect, ('ba', 'abcdefg', 'abdst', 'albmcnd'), False)
['a', 'b']
['a', 'b']
['a', 'b']
['a', 'b']

```

Mieszanie argumentów tutaj nie generuje wszystkich możliwych kolejności argumentów (wymagałoby to wykonania pełnej permutacji i 24 iteracji mieszania dla 4 argumentów), ale wystarczy sprawdzić, czy kolejność argumentów wpływa tutaj na wyniki. Jeżeli będziesz kontynuować testowanie, zauważysz, że *duplikaty* nie pojawią się ani w części wspólnej, ani w sumie, co z perspektywy matematycznej kwalifikuje je jako operacje na zbiorach:

```

>>> intersect([1, 2, 1, 3], (1, 1, 4))
[1]
>>> union([1, 2, 1, 3], (1, 1, 4))
[1, 2, 3, 4]
>>> tester(intersect, ('ababa', 'abcdefga', 'aaaab'), False)
['a', 'b']
['a', 'b']
['a', 'b']

```

Oczywiście z punktu widzenia algorytmu są one nadal dalekie od optymalnych, ale z uwagi na informacje, jakie znajdziesz w ramce poniżej, pozostawimy Ci dalsze ulepszanie tego kodu jako sugerowane do samodzielnego wykonania. Zauważ również, że mieszanie argumentów, takie jak w naszej funkcji, może być ogólnie użytecznym narzędziem, a sama funkcja `tester` byłaby prostsza, gdybyśmy przenieśli mechanizm mieszania do innej funkcji, która mogłaby dowolnie tworzyć lub generować kombinacje argumentów według własnego uznania:

```
>>> def tester(func, items, trace=True):
```

```
for args in scramble(items):
    ...przetwarzanie argumentów...
```

W praktyce tak właśnie uczymy — powrócimy do tego przykładu i rozbudujemy go w rozdziale 20., w którym nauczmy się tworzyć *generatory* zdefiniowane przez użytkownika. Zestaw naszych operacji na zbiorach po raz ostatni zmodyfikujemy w rozdziale 32. i użyjemy w rozwiązaniu jednego z ćwiczeń z części VI jako *klasy* rozszerzającej obiekt listy dodatkowymi metodami.



Powinienem podkreślić, że ponieważ Python zawiera teraz odrębny *typ reprezentujący obiekty zbiorów* (opisany w rozdziale 5.), żaden z omawianych przez nas przykładów przetwarzania zbiorów nie jest już potrzebny. Dołączyciemy je do naszej książki wyłącznie jako przykłady ilustrujące różne techniki tworzenia kodu. A tak na marginesie, ze względu na fakt, że Python jest stale poprawiany i rozbudowywany, mam pewne podejrzenia, że jego deweloperzy w nieco złośliwy sposób knują za moimi plecami, robiąc wszystko, aby przykłady z mojej książki stały się przestarzałe i niepotrzebne!

Emulacja funkcji print z Pythona 3.0

Kończąc niniejszy rozdział, przyjrzymy się ostatniemu przykładowi działania dopasowywania argumentów. Kod zaprezentowany niżej został utworzony z myślą o wykorzystaniu w Pythonie 2.x oraz wersjach wcześniejszych (działa on również w wersji 3.x, jednak używanie go tam nie ma większego sensu). Wykorzystuje on krotkę z dowolnymi argumentami pozycyjnymi `*args` oraz słownik z dowolnymi argumentami opartym na słowach kluczowych `**args` do symulowania tego, co robi funkcja `print` z Pythona 3.x. W zasadzie deweloperzy Pythona mogli w wersji 3.x zaoferować taki kod jako opcję, zamiast całkowicie usuwać wersję 2.x funkcji `print`, ale jednak w wersji 3.x zamiast tego wybrali całkowite zerwanie z przeszłością.

Jak wiemy z rozdziału 11., nie jest to tak naprawdę wymagane, ponieważ programiści mogą w Pythonie 2.x włączyć obsługę funkcji `print` z wersji 3.x za pomocą instrukcji `import` o poniższej postaci (dostępne w wersjach 2.6 i 2.7):

```
from __future__ import print_function
```

W celu zademonstrowania dopasowywania argumentów plik `print3.py` wykonuje to samo działanie za pomocą niewielkiej ilości kodu, który można wykorzystać ponownie; program tworzy ciąg znaków przeznaczony do wyświetlenia i następnie przepuszcza go przez argumenty konfiguracyjne:

```
#!/usr/bin/python
"""

Emulacja działania funkcji 'print' z Pythona 3.x w celu wykorzystania jej w
wersji 2.x (ale działa również w 3.x)

składnia wywołania: print3(*args, sep=' ', end='\n', file=sys.stdout)
"""

import sys

def print3(*args, **kargs):
    sep = kargs.get('sep', ' ')                      # Wartości domyślne argumentów-
    słów kluczowych
```

```

end = kargs.get('end', '\n')
file = kargs.get('file', sys.stdout)
output = ''
first = True
for arg in args:
    output += ('' if first else sep) + str(arg)
    first = False
file.write(output + end)

```

W celu przetestowania działania tego kodu należy zaimportować go do innego pliku bądź do sesji interaktywnej i użyć podobnie jak funkcji `print` z Pythona 3.x. Poniżej znajduje się skrypt testowy `testprint3.py` (warto pamiętać, że funkcja musi nosić nazwę `print3`, ponieważ `print` jest w wersji 2.x słowem zarezerwowanym):

```

from print3 import print3
print3(1, 2, 3)
print3(1, 2, 3, sep='')                                # Wstrzymanie separatora
print3(1, 2, 3, sep='... ')
print3(1, [2], (3,), sep='... ')                         # Różne typy obiektów
print3(4, 5, 6, sep='', end='')                          # Wstrzymanie nowego
                                                       # wiersza
print3(7, 8, 9)
print3()                                                 # Dodanie nowego wiersza
                                                       # (lub pustego wiersza)
import sys
print3(1, 2, 3, sep='??', end='.\n', file=sys.stderr) # Przekierowanie do
                                                       # pliku

```

Po wykonaniu tego kodu w wersji 2.x otrzymujemy te same wyniki co dla funkcji `print` Pythona 3.x:

```
C:\code> c:\python27\python testprint3.py
1 2 3
123
1...2...3
1...[2]...(3,)
4567 8 9
1??2??3.
```

Choć nie ma to większego znaczenia, po wykonaniu kodu w Pythonie 3.x wyniki są identyczne. Jak zawsze uniwersalność projektu Pythona pozwala nam na tworzenie prototypów i rozwijanie koncepcji w samym języku Python. W tym przypadku narzędzia służące do dopasowywania argumentów są w kodzie napisanym w Pythonie tak samo elastyczne jak w wewnętrznej implementacji tego języka.

Wykorzystywanie argumentów ze słowami kluczowymi

Ciekawostką jest, że powyższy przykład można utworzyć z wykorzystaniem opisanych wcześniej w niniejszym rozdziale argumentów ze słowami kluczowymi z Pythona 3.x, służących do automatycznego sprawdzania poprawności argumentów konfiguracyjnych. Kod źródłowy przykładu przedstawionego poniżej znajdziesz w pliku *print3_alt1.py*:

```
#!/usr/bin/python3

# Używamy tylko argumentów 3.x ze słowami kluczowymi

import sys

def print3(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

Ta wersja kodu działa tak samo jak oryginał i jest świetnym przykładem użyteczności argumentów będących słowami kluczowymi. W oryginalnej wersji kodu zakładamy, że wszystkie argumenty pozycyjne mają być wyświetlane, natomiast wszystkie argumenty ze słowami kluczowymi służą do podania opcji. To właściwie wystarczy, jednak wszelkie dodatkowe argumenty-słowa kluczowe będą po cichu ignorowane. Wywołanie podobne do poniższego wygeneruje na przykład wyjątek w przypadku zastosowania formy z argumentem mogącym być tylko słowem kluczowym:

```
>>> print3(99, name='robert')
TypeError: print3() got an unexpected keyword argument 'name'
```

natomiast w oryginalnej wersji argument *name* zostanie po cichu zignorowany. W celu ręcznego wykrywania zbędnych słów kluczowych moglibyśmy użyć *dict.pop()* i usunąć pobrane wpisy, a następnie sprawdzić, czy słownik nie jest pusty. Kolejna wersja, dostępna w pliku *print3_alt2.py*, jest równoważna wersji zawierającej tylko słowa kluczowe — zgłasza wbudowany wyjątek za pomocą polecenia *raise*, które działa tak, jak gdyby to zrobił Python (przyjrzymy się temu bardziej szczegółowo w części VII):

```
#!/usr/bin/python

"Używamy tylko argumentów ze słowami kluczowymi 2.x/3.x z wartościami
domyślnymi"

import sys

def print3(*args, **kargs):
    sep = kargs.pop('sep', ' ')
    end = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError('dodatkowe słowa kluczowe: %s' % kargs)
```

```

output = ''
first = True
for arg in args:
    output += ('' if first else sep) + str(arg)
    first = False
file.write(output + end)

```

Powyższy kod działa jak poprzednia wersja, jednak przechwytuje teraz również dodatkowe argumenty będące słowami kluczowymi:

```

>>> print3(99, name='robert')
TypeError: dodatkowe słowa kluczowe: {'name': 'robert'}

```

Ta wersja funkcji działa w Pythonie 2.x, jednak wymaga o cztery wiersze kodu więcej od wersji z argumentami będącymi słowami kluczowymi. Niestety, dodatkowe wiersze kodu są w tym przypadku niezbędne — wersja z argumentami w postaci słów kluczowych działa jedynie w Pythonie 3.x, co stoi w sprzeczności z powodami tworzenia tego przykładu (emulator wersji 3.x działający jedynie w wersji 3.x nie jest szczególnie przydatny!). W programach pisanych z myślą o Pythonie 3.x argumenty mogące być jedynie słowami kluczowymi mogą uprościć pewną kategorię funkcji, które przyjmują zarówno argumenty, jak i opcje. Kolejny przykład działania argumentów będących słowami kluczowymi znajduje się w rozdziale 21., w studiu przypadku pomiaru czasu wykonywania iteracji.

Warto pamiętać: argumenty ze słowami kluczowymi

Jak mogłeś się sam przekonać, zaawansowane tryby dopasowywania argumentów mogą być zagadnieniem dosyć złożonym. Są również całkowicie opcjonalne. Można sobie zupełnie dobrze radzić, stosując proste dopasowanie pozycyjne, i najprawdopodobniej szczególnie na początku jest to niezły pomysł. Ponieważ jednak niektóre narzędzia Pythona z nich korzystają, dobrze jest mieć choć ogólną wiedzę na temat trybów specjalnych.

Na przykład argumenty ze słowami kluczowymi odgrywają szczególnie istotną rolę w interfejsie tkinter, będącym de facto standardowym API graficznego interfejsu użytkownika w Pythonie (w Pythonie 2.x moduł ten nosi nazwę Tkinter). W różnych miejscach książki krótko wspominamy o module tkinter; jeżeli chodzi o jego wzorce wywoływania, argumenty ze słowami kluczowymi ustawiają opcje konfiguracyjne przy budowaniu komponentów GUI. Przykładowo wywołanie w formie:

```

from tkinter import *
widget = Button(text="Naciśnij mnie", command=someFunction)

```

tworzy nowy przycisk i określa jego etykietę oraz funkcję zwrotną, używając do tego argumentów ze słowami kluczowymi text oraz command. Ponieważ liczba opcji konfiguracji dla widgetu może być spora, argumenty ze słowami kluczowymi pozwalają nam wybierać z nich odpowiednie. Bez tego albo trzeba wymienić listę wszystkich możliwych opcji zgodnie z ich pozycją, albo liczyć na rozsądny protokół pozycyjnych wartości domyślnych, który poradzi sobie z każdym możliwym ułożeniem opcji.

Wiele funkcji wbudowanych Pythona oczekuje, że w różnych trybach użycia przekażemy im słowa kluczowe jako opcje, które mogą mieć wartości domyślne, ale nie muszą. Jak wiemy z rozdziału 8., funkcja wbudowana sorted w postaci:

```
sorted(iterable, key=None, reverse=False)
```

oczekuje przekazania do sortowania obiektu, na którym można wykonać iterację, jednak pozwala nam także przekazać opcjonalne argumenty-słowa kluczowe w celu wybrania

klucza sortowania słownika oraz opcji odwrócenia kierunku sortowania — domyślnie mają one wartości, odpowiednio, `None` oraz `False`. Ponieważ normalnie nie korzystamy z tych opcji, można je pominąć w celu zastosowania wartości domyślnych.

Jak widzieliśmy, wywołania `dict`, `str.format` i `print` z wersji 3.x również akceptują słowa kluczowe — są to zastosowania, które musielibyśmy wprowadzić we wcześniejszych rozdziałach z powodu ich zależności od trybów przekazywania argumentów omawianych tutaj (niestety ci, którzy zmieniają Pythona, znają już Pythona!).

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy drugą kluczową koncepcję związaną z funkcjami — *argumenty* (sposoby przekazywania obiektów do funkcji). Jak już wiemy, argumenty przekazywane są do funkcji przez przypisanie, co oznacza referencję do obiektu, a tak naprawdę wskaźnik. Omówiliśmy także bardziej zaawansowane rozszerzenia — na przykład argumenty ze słowami kluczowymi oraz wartościami domyślnymi, narzędzia służące do wykorzystywania dowolnej liczby argumentów, a także argumenty mogące być tylko słowami kluczowymi z wersji 3.x. Wreszcie widzieliśmy, jak zmienne argumenty mogą przejawiać takie samo zachowanie jak inne współdzielone referencje do obiektów — o ile obiekt nie jest kopowany w sposób jawnym przy przesyłaniu, modyfikacja przekazanego obiektu zmennego może wpływać na kod wywołujący.

W kolejnym rozdziale kontynuujemy omówienie funkcji, zgłębiając bardziej zaawansowane zagadnienia — adnotacje funkcji, wyrażenia `lambda`, rekurencję oraz narzędzia funkcyjne, takie jak `map` i `filter`. Wiele z tych koncepcji wynika z faktu, że funkcje są w Pythonie normalnymi obiektami, dlatego obsługują pewne zaawansowane i bardzo elastyczne tryby przetwarzania. Przed przejściem do tych zagadnień czas zajść się quizem sprawdzającym związane z argumentami kwestie omówione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

W większości pytań tego quizu wyniki mogą się nieznacznie różnić w wersji 2.x — mogą zawierać dodatkowe nawiasy i przecinki, zwłaszcza gdy wyświetlanych jest wiele wartości. Aby w wersji 2.x wyświetlać dane dokładnie w taki sposób, jak robi to wersja 3.x, przed rozpoczęciem zaimportuj funkcję `print_function` z modułu `_future_`.

1. Jaki będzie wynik działania poniższego kodu i dlaczego?

```
>>> def func(a, b=4, c=5):
    print(a, b, c)
```

```
>>> func(1, 2)
```

2. Jaki będzie wynik działania poniższego kodu i dlaczego?

```
>>> def func(a, b, c=5):
    print(a, b, c)
```

```
>>> func(1, c=3, b=2)
```

3. Jakie dane wyjściowe zwraca poniższy kod i dlaczego?

```
>>> def func(a, *pargs):
    print(a, pargs)
```

```
>>> func(1, 2, 3)
```

4. Co wyświetla poniższy kod i dlaczego?

```
>>> def func(a, **kargs):
    print(a, kargs)
```

```
>>> func(a=1, c=3, b=2)
```

5. Jaki będzie wynik działania poniższego kodu i dlaczego?

```
>>> def func(a, b, c=3, d=4): print(a, b, c, d)
```

```
>>> func(1, *(5,6))
```

6. I jeszcze jeden, ostatni raz: jaki będzie wynik działania poniższego kodu i dlaczego?

```
>>> def func(a, b, c): a = 2; b[0] = 'x'; c['a'] = 'y'
>>> l=1; m=[1]; n={'a':0}
>>> func(l, m, n)
>>> l, m, n
```

Sprawdź swoją wiedzę – odpowiedzi

1. Wynikiem będzie '1 2 5', ponieważ 1 i 2 przekazywane są do a i b przez pozycję, natomiast c jest pominięte w wywołaniu i otrzymuje wartość domyślną 5.
2. Wynikiem tym razem będzie '1 2 3'. 1 przekazywane jest do a przez pozycję, natomiast do b oraz c przez nazwę przekazane zostają 2 i 3 (kolejność od lewej do prawej strony nie ma znaczenia, kiedy w taki sposób wykorzystujemy argumenty-słowa kluczowe).
3. Kod wyświetla '1 (2, 3)', ponieważ 1 przekazywane jest do a, natomiast *pargs zbiera pozostałe argumenty pozycyjne w nowy obiekt krotki. Krotkę z dodatkowymi argumentami pozycyjnymi możemy przejść za pomocą dowolnego narzędzia iteracyjnego (na przykład `for arg in pargs: ...`).
4. Tym razem kod wyświetla '1, {'b': 2, 'c': 3}', ponieważ 1 przekazane zostaje do a przez nazwę, natomiast **kargs zbiera pozostałe argumenty-słowa kluczowe w słowniku. Słownik z dodatkowymi argumentami-słowami kluczowymi możemy przejść po kluczu za pomocą dowolnego narzędzia iteracyjnego (na przykład `for key in kargs: ...`). Pamiętaj, że kolejność elementów słownika może się różnić w zależności od wersji Pythona i innych zmiennych.

5. Wynikiem będzie tym razem '1 5 6 4'. 1 dopasowane zostaje do a za pomocą pozycji, 5 i 6 dopasowane zostają do b i c za pomocą konstrukcji **nazwa* (6 nadpisuje wartość domyślną c), natomiast d ma wartość domyślną 4, ponieważ do zmiennej tej nie przekazano wartości.
6. Wynikiem działania będzie (1, ['x'], {'a': 'y'}) — pierwsze przypisanie w funkcji nie wpływa na kod wywołujący, ale dwa pozostałe tak, ponieważ zmieniają w miejscu przekazywane obiekty mutowalne.

[1] Tak naprawdę jest to dość skomplikowane. Procedura `sort` w Pythonie została napisana w języku C i wykorzystuje bardzo zoptymalizowany algorytm, który próbuje korzystać z częściowego uporządkowania elementów do sortowania. Nosi on nazwę *timsort* od imienia twórcy, Tim'a Petersa, i zgodnie z dokumentacją ma w niektórych sytuacjach mieć „ponadnaturalną wydajność” (całkiem nieźle, jak na sortowanie!). Mimo to sortowanie to operacja potencjalnie wykładnicza (musi wiele razy pociąć sekwencję na kawałki i ponownie ją złożyć), a inne wersje rozwiązania po prostu wykonują jedno liniowe przeglądanie kolekcji argumentów. W rezultacie sortowanie jest szybsze, jeśli argumenty są częściowo posortowane, jednak w innym przypadku prawdopodobnie będzie wolniejsze (co się po raz kolejny potwierdziło w testach przeprowadzonych w wersji 3.3). Wydajność Pythona może jednak z czasem się zmieniać, a sam fakt, że sortowanie zaimplementowane jest w języku C, może znacznie pomóc. By dokonać dokładnej analizy, powinniśmy zmięczyć czas działania alternatywnych rozwiązań za pomocą modułów `time` i `timeit` omówionych w rozdziale 21.

Rozdział 19. Zaawansowane zagadnienia dotyczące funkcji

Niniejszy rozdział wprowadza wiele bardziej zaawansowanych zagadnień związanych z funkcjami — funkcje rekurencyjne, atrybuty i adnotacje, wyrażenie `lambda`, narzędzi programowania funkcyjnego, takie jak `map` i `filter`. To bardziej zaawansowane narzędzia, których część Czytelników może nie napotkać w swojej codziennej pracy. Jednak z powodu ich znaczenia w niektórych dziedzinach programowania użytkownika może okazać się ich znajomość choćby na podstawowym poziomie. Na przykład konstrukcje `lambda` są dość powszechnie w programowaniu graficznych interfejsów użytkownika, a techniki programowania funkcyjnego są coraz bardziej popularne w kodzie Pythona.

Część sztuki wykorzystywania funkcji polega na używaniu interfejsów pomiędzy nimi, dlatego przy okazji omówimy również pewne ogólne zasady projektowania funkcji. Następny rozdział kontynuuje to zagadnienie, wprowadzając zaawansowane pojęcia funkcji i wyrażenia generatorów oraz pogłębiając tematykę list składanych i ich zastosowań w kontekście narzędzi funkcyjnych przedstawionych w niniejszym rozdziale.

Koncepcje projektowania funkcji

Skoro poznaliśmy już podstawy tworzenia funkcji w Pythonie, warto powiedzieć parę słów na temat kontekstu. Gdy ktoś zaczyna na poważnie korzystać z funkcji, powinien podjąć kilka decyzji dotyczących kontekstu, na przykład sposobu rozbijania zadania na funkcje (co jest znane pod pojęciem *spójności*, ang. *cohesion*), tego, w jaki sposób funkcje powinny komunikować się wzajemnie (co jest określone jako *sprzęganie*, ang. *coupling*) i tak dalej. Należy również wziąć pod uwagę takie rzeczy, jak rozmiary funkcji, ponieważ ten czynnik ma bezpośredni wpływ na użyteczność kodu. Niektóre z tych zagadnień należą do kategorii analizy strukturalnej i projektowania i mają zastosowanie do kodu w Pythonie tak samo jak do innych języków programowania.

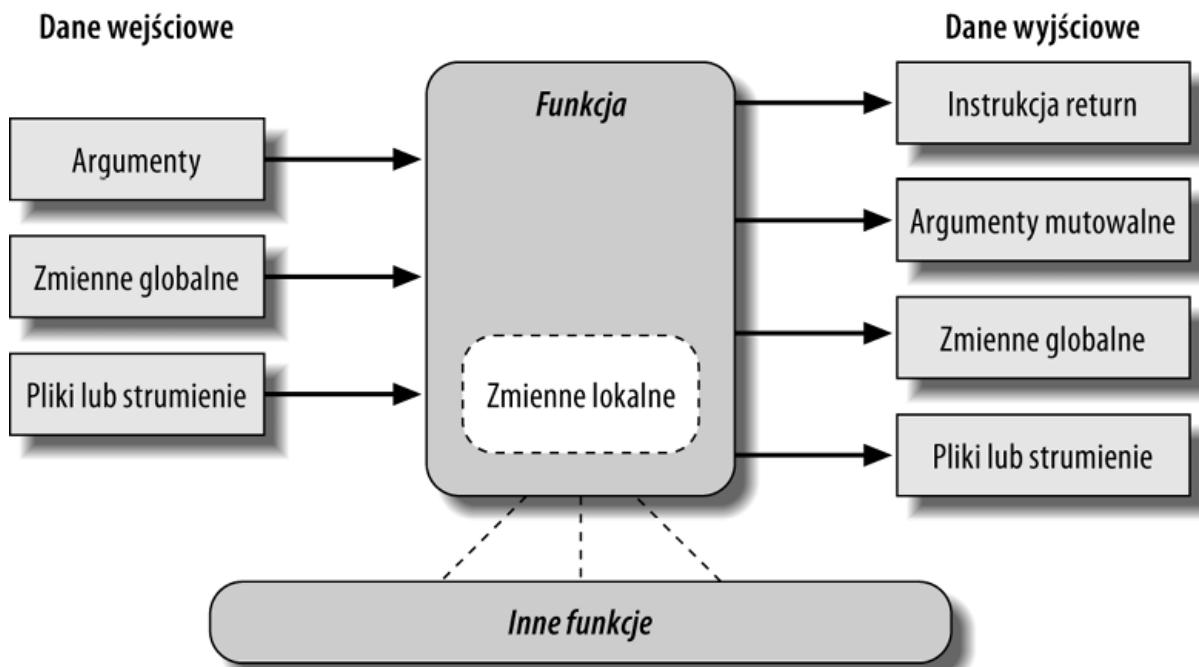
Niektóre zagadnienia dotyczące sprzęgania w kontekście funkcji i modułów poznaliśmy w rozdziale 17. przy okazji zakresów, poniżej przedstawiam krótkie podsumowanie tych zagadnień dla początkujących:

- **Sprzęganie: używanie argumentów jako danych wejściowych oraz instrukcji `return` do zwracania wyników.** Należy unikać uzależniania funkcji od jej otoczenia. Wykorzystanie argumentów i instrukcji `return` to najlepszy sposób odizolowania funkcji od zależności zewnętrznych, przy pozostawieniu ustalonych, dobrze oznaczonych punktów styku.
- **Sprzęganie: używanie zmiennych globalnych wyłącznie tam, gdzie to konieczne.** Zmienne globalne (to znaczy nazwy w module zawierającym funkcję) są z reguły niezalecanym sposobem komunikacji z funkcją. Mogą powodować skomplikowane zależności i inne trudne do wyśledzenia problemy z debugowaniem, modyfikowaniem i ponownym wykorzystywaniem kodu.
- **Sprzęganie: nie należy modyfikować argumentów mutowalnych, o ile kod wywołujący tego nie oczekuje.** Funkcje mogą modyfikować część przekazywanych im obiektów mutowalnych, ale to może powodować mnóstwo sprzężeń między kodem

wywołującym a wywoływanym, co z kolei powoduje, że funkcje stają się bardzo zależne od otoczenia.

- **Spójność:** każda funkcja powinna mieć jeden, zunifikowany cel. W przypadku dobrego projektu każda z funkcji powinna wykonywać pojedyncze zadanie, czynność, którą można określić jednym zdaniem oznajmującym. Jeśli to zdanie jest bardzo ogólne (na przykład: „Ta funkcja implementuje całą logikę programu”) lub złożone (na przykład: „Funkcja podnosi pracownikowi pensję i składa zamówienie na pizzę”), warto zastanowić się nad rozbiciem funkcji na kilka mniejszych. W przeciwnym razie ponowne użycie funkcji może okazać się trudne z powodu skomplikowanego kontekstu jej zastosowania.
- **Rozmiar:** każda funkcja powinna być relatywnie mała. Ten wymóg wynika z poprzedniego, ale warto zwrócić uwagę na to, że jeśli funkcja zajmuje kilka ekranów na wyświetlaczu, to dobry sygnał, że warto ją rozbić na mniejsze części. Szczególnie biorąc pod uwagę spójność i czytelność jako istotniejsze cechy Pythona, rozlewka i głęboko zagnieżdżona logika funkcji jest często symptomem problemów projektowych. Klucz do sukcesu leży w prostocie i zwięzłości.
- **Sprzęganie:** należy unikać bezpośredniego modyfikowania zmiennych zdefiniowanych w innym pliku modułu. Tę koncepcję wprowadziliśmy w rozdziale 17. i pogłębimy ją w następnej części książki, przy okazji modułów. Jednak w celu zachowania kompletności tego wyliczenia należy wspomnieć o tym, że modyfikowanie zmiennych zdefiniowanych w innych modułach powoduje sprzężenie między modułami, analogiczne do sprzężenia między funkcjami, powodowanego przez użycie zmiennych globalnych. Powoduje, że moduły stają się skomplikowane i trudne do ponownego użycia. Wszędzie, gdzie to możliwe, należy stosować funkcje akcesorów zamiast bezpośrednich instrukcji przypisania.

Rysunek 19.1 zawiera podsumowanie metod komunikacji funkcji z ich otoczeniem: elementy po lewej stronie to możliwe źródła danych wejściowych, wyniki są pokazane po prawej. Dobry projekt funkcji wykorzystuje wyłącznie przekazywanie danych wejściowych w postaci argumentów, a zwracanie wyników za pomocą instrukcji return.



Rysunek 19.1. Środowisko wykonawcze funkcji. Funkcje mogą uzyskiwać dane wejściowe i zwracać wyniki na wiele sposobów, ale z reguły najłatwiej zrozumieć i rozwijać funkcje

pobierające dane wejściowe wyłącznie z argumentów, a zwracające wyniki za pomocą instrukcji `return`, względnie w efekcie modyfikacji mutowalnych argumentów. W Pythonie 3.x wynikiem funkcji mogą być również zadeklarowane zmienne nonlocal istniejące w zasięgu definicji funkcji

Oczywiście istnieje mnóstwo wyjątków od przedstawionych wyżej reguł, między innymi wynikających z technik programowania obiektowego. Jak zobaczymy w części VI, funkcjonowanie klas w Pythonie opiera się na modyfikowaniu instancji klasy przekazywanej do jej metod, innymi słowy, funkcje zdefiniowane w ciele klasy modyfikują przekazywany do nich automatycznie obiekt instancji `self` (na przykład w efekcie przypisania typu `self.name = 'bob'`). Co więcej, jeżeli klasy nie są używane, najprostszym i najpowszechniejszym sposobem przekazywania stanu między wywołaniami funkcji są zmienne globalne. Efekty uboczne są niebezpieczne wyłącznie wówczas, gdy nie są spodziewane.

Jednak w ujęciu ogólnym należy starać się minimalizować zewnętrzne zależności funkcji i innych elementów programów. Im bardziej funkcja będzie *niezależna od otoczenia*, tym będzie łatwiejsza do zrozumienia, ponownego użycia i modyfikacji.

Funkcje rekurencyjne

O rekurencji wspominaliśmy już przy okazji porównywania typów w rozdziale 9. Z kolei przy okazji omawiania reguł zasięgów na początku rozdziału 17. mówiliśmy o tym, że Python obsługuje *funkcje rekurencyjne*, to znaczy funkcje, które wywołują same siebie bezpośrednio lub za pośrednictwem innych funkcji. W tej sekcji opowiemy o tym, jak to wygląda w kodzie naszych funkcji.

Rekurencja jest dość zaawansowanym zagadnieniem informatycznym i w Pythonie spotykamy ją dość rzadko, po części dlatego, że instrukcje proceduralne Pythona zawierają prostsze struktury pętli. Nie zmienia to jednak w niczym faktu, że jest to jednak użyteczna technika i warto o niej wiedzieć, ponieważ pozwala programistom przetwarzanie danych cechujące się trudną do przewidzenia strukturą i zagnieżdżeniem — przykładami takich zadań mogą być planowanie tras podróży, analizy językowe czy przeglądanie wszystkich łączy danej witryny internetowej. Rekurencja bywa czasem alternatywą dla prostych pętli i iteracji, choć niekoniecznie musi to być alternatywa czytelniejsza lub łatwiejsza do zrozumienia.

Sumowanie z użyciem rekurencji

Przyjrzyjmy się kilku przykładom. Aby policzyć sumę elementów listy (lub innej sekwencji), można użyć wbudowanej funkcji `sum` lub napisać jej własną wersję. Gdyby taką funkcję napisać z użyciem rekurencji, miałaby mniej więcej taką postać:

```
>>> def mysum(L):
    if not L:
        return 0
    else:
        return L[0] + mysum(L[1:]) # Wywołanie samej
                                     siebie(rekurencja)
>>> mysum([1, 2, 3, 4, 5])
```

Przy każdym wywołaniu funkcja rekurencyjnie wywołuje samą siebie w celu policzenia sumy pozostały elementów listy, po czym ta suma jest dodawana do pierwszego odczytanego elementu. Rekurencyjne wywołania kończą się, gdy podlista jest pusta — w tym przypadku funkcja zwraca zero. W tego typu wywołaniu rekurencyjnym każdy poziom zagnieźdżenia rekurencyjnego otrzymuje własną kopię lokalnego zakresu funkcji, co w naszym przykładzie oznacza, że L jest inną zmienną dla każdego wywołania.

Jeśli ten przykład jest trudny do zrozumienia (co się często zdarza w przypadku początkujących programistów), można do funkcji dodać wyświetlanie wartości L i wywołać ją ponownie, dzięki czemu będziemy mieli wgląd w wartość tej zmiennej na każdym poziomie wywołania.

```
>>> def mysum(L):
    print(L)                                     # Śledzenie poziomów
    rekurencji
    if not L:                                    # L z każdym wywołaniem ma
        coraz mniejszą długość
    return 0
else:
    return L[0] + mysum(L[1:])
>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Jak widać, sumowana lista zmniejsza się dla każdego kolejnego zagnieźdżenia rekurencyjnego, aż staje się pusta, co powoduje zakończenie sekwencji rekurencyjnej. Suma jest obliczana wraz z *odwijaniem* się rekurencji.

Implementacje alternatywne

Co interesujące, alternatywnie możemy użyć trójskładnikowej operacji `if/else` (opisanej w rozdziale 12.), dzięki czemu możemy oszczędzić nieco kodu. Możemy również uogólnić funkcję, pozwalając zastosować listy dowolnych elementów, które można sumować (co okaże się łatwiejsze, jeśli przyjmiemy założenie, że dane wejściowe zawierają przynajmniej jeden element, podobnie jak to miało miejsce w rozdziale 18. w przykładzie wyznaczającym wartość minimalną). Zastosujemy rozszerzoną składnię przypisania sekwencyjnego wprowadzoną w 3.x, dzięki czemu rozpakowanie wartości będzie prostsze (szczegółowy opis tej techniki znajduje się w rozdziale 11.).

```
def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:])           # Użycie wyrażeń
trójskładnikowych
def mysum(L):
```

```

    return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Dowolny typ,
domyślnie wartość 1

def mysum(L):
    first, *rest = L

    return first if not rest else first + mysum(rest)      # Użycie
rozszerzonego przypisania

                                            # sekwencji z 3.x

```

Zastosowane techniki nie obsługują pustych danych wejściowych, ale w zamian za to możemy użyć danych dowolnego typu obsługującego operator dodawania (+), a więc nie jesteśmy już ograniczeni do liczb.

```

>>> mysum([1])                                # mysum([]) nie zadziała dla
ostatnich 2 wersji funkcji

1

>>> mysum([1, 2, 3, 4, 5])
15

>>> mysum(('j', 'a', 'j', 'k', 'o'))          # Ale funkcja działa dla różnych
typów

'jajko'

>>> mysum(['mielonka', 'szynka', 'jajka'])
'mielonkaszynkajajka'

```

Spróbuj samodzielnie poeksperymentować z tymi przykładami, aby lepiej zrozumieć ich działanie. Jeżeli dokładnie przeanalizujesz te trzy warianty, z pewnością zauważysz, że:

- Dwa ostatnie warianty obsługują również pojedynczy argument będący ciągiem znaków (na przykład `mysum('mielonka')`), ponieważ ciągi znaków są sekwencjami pojedynczych znaków.
- Trzeci wariant obsługuje dodatkowo dowolne obiekty iterowalne, z otwartymi plikami (np. `mysum(open(nazwa))`) włącznie, ale dwa pierwsze już nie, ponieważ wykorzystują dostęp do elementów po indeksie (w rozdziale 14. znajdziesz więcej szczegółowych informacji na temat rozszerzonych przypisań sekwencji do plików).
- Nagłówek funkcji `def mysum(first, *rest)`, choć podobny do trzeciego wariantu, nie będzie działał, ponieważ funkcja oczekuje indywidualnych elementów, a nie pojedynczego obiektu iterowalnego.

Powinieneś pamiętać, że rekurencja może być bezpośrednia, jak w przedstawionych przykładach, lub *pośrednia*, jak w listingu poniżej (funkcja wywołuje inną funkcję, która z kolei wywołuje tę pierwszą). Efekt jest ten sam, ale na każdym poziomie rekurencji mamy wtedy do czynienia z dwiema funkcjami, a nie z jedną.

```

>>> def mysum(L):
    if not L: return 0
    return nonempty(L)                      # Wywołuję funkcję, która
wywołała mnie

>>> def nonempty(L):
    return L[0] + mysum(L[1:])            # Rekurencja pośrednia

```

```
>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

Pętle a rekurencja

Rekurencja działa prawidłowo w przykładach prezentowanych w poprzednim punkcie, ale jej zastosowanie w tak prostym zadaniu to przerost formy nad treścią. W rzeczywistości rekurencja nie jest stosowana w Pythonie tak często jak w innych językach programowania, na przykład w Prologu czy Lisp, ponieważ Python kładzie nacisk na prostsze instrukcje proceduralne, jak pętle, które z reguły pozwalają uzyskać znacznie bardziej naturalniejsze rozwiązania. Na przykład pętla `while` często oferuje bardziej konkretny algorytm i nie wymaga pisania funkcji rekurencyjnej.

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> while L:
    sum += L[0]
    L = L[1:]
>>> sum
15
```

Co więcej, pętla `for` automatycznie wykonuje iterację, dzięki czemu w większości przypadków rekurencja staje się zupełnie zbędna (a co nie mniej ważne, najczęściej mniej wydajna z punktu widzenia zużycia pamięci i czasu wykonania).

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> for x in L: sum += x
>>> sum
15
```

W przypadku pętli nie potrzebujemy kopii fragmentu sekwencji w lokalnym zakresie funkcji dla każdej iteracji, unikamy też kosztów czasowych związanych z wywołaniami funkcji (w rozdziale 21. poświęcimy nieco więcej miejsca na zagadnienia związane z pomiarem wydajności alternatywnych implementacji).

Obsługa dowolnych struktur

Z drugiej strony, rekurencja (albo analogiczne algorytmy wykorzystujące stos danych, z którymi spotkamy się już niebawem) może być wykorzystana do przeglądania danych o dowolnie złożonych strukturach. Jako prostym przykładem rekurencji wykorzystanej w takim kontekście posłużymy się zadaniem obliczenia sumy elementów zapisanych w wielokrotnie zagnieżdżonych listach:

```
[1, [2, [3, 4], 5], 6, [7, 8]]                      # Dowolnie zagnieżdżone
podlisty
```

Prosta pętla nie poradzi sobie z takim zadaniem, ponieważ nie mamy tutaj do czynienia z liniową iteracją. Zagnieżdżone pętle również na nic się zdadzą, ponieważ podlisty mogą być dowolnie zagnieżdżone, dając w efekcie dowolną strukturę — w takim scenariuszu nie ma możliwości wcześniejszego przewidzenia tego, jak wiele zagnieżdżonych pętli będziemy musieli wykonać. Zamiast pętlą posłużymy się więc następującym kodem wykorzystującym rekurencję do analizowania napotkanych po drodze zagnieżdżonych podlist.

```
#plik sumtree.py

def sumtree(L):
    tot = 0
    for x in L:
        # Dla każdego elementu na tym poziomie
        if not isinstance(x, list):
            tot += x
            # Bezpośrednie dodawanie liczb
        else:
            tot += sumtree(x)
            # Rekurencja dla podlist
    return tot

L = [1, [2, [3, 4], 5], 6, [7, 8]]
print(sumtree(L))
# Dowolne zagnieżdżanie
# Wypisuje 36

# Przypadki zdegenerowane
print(sumtree([1, [2, [3, [4, [5]]]]]))      # Wypisuje 15 (drzewo
# rozrośnięte po prawej)
print(sumtree([[[[[1, 2], 3], 4], 5]]))       # Wypisuje 15 (drzewo
# rozrośnięte po lewej)
```

Warto prześledzić dwa ostatnie przykłady wywołania, aby zrozumieć, w jaki sposób rekurencja przeszukuje zagnieżdżone listy.

Rekurencja kontra kolejki i stosy

Zrozumienie sposobu działania rekurencji może być łatwiejsze, jeżeli uświadomisz sobie, że wewnętrznie Python implementuje rekurencję poprzez umieszczanie informacji o stanie na stosie wywołań przy każdym wywołaniu rekurencyjnym, dzięki czemu pamięta, gdzie musi wrócić i jak kontynuować działanie po powrocie. W rzeczywistości możliwe jest wdrożenie procedur w stylu rekurencyjnym działających bez wywołań rekurencyjnych; możemy to zrobić za pomocą własnego stosu lub kolejki, pozwalających na śledzenie bieżących działań i kroków pozostałych do wykonania.

Na przykład funkcja, której kod źródłowy przedstawiamy poniżej, oblicza te same sumy, co w poprzednim przykładzie, ale zamiast rekurencji używa jawniej listy do zaplanowania, kiedy zostaną odwiedzone kolejne elementy. Element znajdujący się na początku listy jest zawsze następnym do przetworzenia i zsumowania:

```
def sumtree(L):                                # Wersja z wykorzystaniem jawniej
    kolejki
    tot = 0
    items = list(L)                             # Zacznię od utworzenia kopii
    najwyższo poziomu
```

```

while items:
    front = items.pop(0)                      # Pobierz / usuń pierwszy element
    if not isinstance(front, list):
        tot += front                          # Dodaj liczby bezpośrednio
    else:
        items.extend(front)                   # <== Dołączaj kolejne elementy na
                                                # końcu listy zagnieżdżonej
    return tot

```

Technicznie rzecz biorąc, kod funkcji przechodzi przez listę według kolejnych poziomów i dodaje zawartość zagnieżdżonych list na końcu głównej listy, tworząc *kolejkę* typu FIFO (ang. *first in, first out*; pierwszy na wejściu, pierwszy na wyjściu). Aby dokładniej emulować rekurencyjne przechodzenie przez kolejne elementy, możemy go zmienić tak, aby w pierwszej kolejności analizowany był najbardziej zagnieżdżony element, a zawartość zagnieżdżonych list dodawana była na początku listy głównej, tworząc *stos* typu FILO (ang. *first in, last out*; pierwszy na wejściu, ostatni na wyjściu):

```

def sumtree(L):                                # Wersja z wykorzystaniem jawnego
    stosu

    tot = 0

    items = list(L)                            # Zacznij od utworzenia kopii
                                                # najwyższego poziomu

    while items:
        front = items.pop(0)                  # Pobierz / usuń pierwszy element
        if not isinstance(front, list):
            tot += front                      # Dodaj liczby bezpośrednio
        else:
            items[:0] = front                 # <== Dołączaj kolejne elementy na
                                                # końcu listy zagnieżdżonej
    return tot

```

Więcej informacji na temat dwóch ostatnich przykładów (i jeszcze innego wariantu) możesz znaleźć w pliku *sumtree2.py* w pakiecie przykładów do książki. W kodzie zapisanym w pliku zaimplementowano również śledzenie listy elementów, dzięki czemu można obserwować, jak rośnie w obu schematach, a także wyświetlanie przetwarzanych elementów odwiedzania, co pozwala analizować kolejność wyszukiwania. Na przykład warianty z kolejką i stosem odwiedzają elementy tych samych trzech list testowych, które zostały użyte dla wersji rekurencyjnej, odpowiednio w następującej kolejności (sumy są pokazane na końcu):

```

c:\code> sumtree2.py

1, 6, 2, 5, 7, 8, 3, 4, 36
1, 2, 3, 4, 5, 15
5, 4, 3, 2, 1, 15
-----
1, 2, 3, 4, 5, 6, 7, 8, 36

```

1, 2, 3, 4, 5, 15

1, 2, 3, 4, 5, 15

Zasadniczo jednak, kiedy zrozumiesz zasady działania wywołań rekurencyjnych, okazują się one bardziej naturalne niż jawne listy czy kolejki i są ogólnie bardziej preferowanym rozwiązaniem, chyba że musimy przetwarzać daną strukturę w ścisłe określony, wyspecjalizowany sposób. Na przykład niektóre programy przeprowadzają wyszukiwanie elementów w sposób, który wymaga zdefiniowanej kolejki wyszukiwania uporządkowanej według *trifności* lub innych kryteriów. Jeżeli wyobrażisz sobie robota sieciowego, który ocenia odwiedzane strony na podstawie ich treści, działanie takich aplikacji stanie się bardziej zrozumiałe.

Cykle, ścieżki i ograniczenia stosu

Jak na razie przedstawione przykłady programów są zupełnie wystarczające dla naszych potrzeb, ale bardziej złożone aplikacje rekurencyjne mogą czasem wymagać nieco więcej infrastruktury, niż tutaj pokazaliśmy: mogą wymagać unikania odwołań cyklicznych lub powtórzeń, zapisywania wykorzystanych ścieżek wyszukiwania do późniejszego użycia czy powiększania przestrzeni stosu podczas używania wywołań rekurencyjnych zamiast jawnych kolejek lub stosu.

Żaden z przykładów wykorzystania wywołań rekurencyjnych czy jawnych implementacji kolejek lub stosu prezentowanych w tym rozdziale nie pozwala na uniknięcie *cykli*, czyli na unikanie odwiedzania miejsc, które już zostały odwiedzone. Nie jest to tutaj co prawda wymagane, ponieważ przemierzamy ścisłe hierarchiczne drzewa obiektów listy. Jeżeli jednak struktura danych będzie grafem cyklicznym, oba te schematy zawiodą: wersja z wywołaniem rekurencyjnym wpadnie w nieskończoną pętlę (i może zabraknąć miejsca na stosie wywołań), a inne rozwiązania wpadną w proste pętle nieskończone, w kółko dodając te same elementy do listy (co może spowodować przepełnienie dostępnej pamięci systemu). Niektóre programy muszą także unikać powtórnego przetwarzania stanów, które już raz zostały osiągnięte, nawet jeżeli nie doprowadziły do powstania niepożądanej pętli.

Aby to zrobić lepiej, wersja z wywołaniem rekurencyjnym może po prostu przechowywać i przekazywać zbiór, słownik lub listę odwiedzonych do tej pory stanów i sprawdzać, czy są powtarzane. Wykorzystamy ten schemat w późniejszych przykładach rekurencyjnych w tej książce:

```
if state not in visited:  
    visited.add(state)                      # x.add(state), x[state]=True, lub  
    x.append(state)  
    ...przetwarzanie...
```

Nierekurencyjne rozwiązania alternatywne mogłyby w podobny sposób unikać dodawania odwiedzonych stanów za pomocą kodu pokazanego poniżej. Zauważ, że sprawdzanie duplikatów znajdujących się już na liście elementów pozwoliłoby uniknąć dwukrotnego zaplanowania przetwarzania danego stanu, ale nie przeszkodziłoby w ponownym odwiedzeniu stanu, który został wcześniej odwiedzony, a zatem usunięty z tej listy:

```
visited.add(front)  
...przetwarzanie...  
items.extend([x for x in front if x not in visited])
```

Ten model nie ma zastosowania do przypadku użycia opisanego w tej sekcji, gdzie po prostu dodajemy liczby do listy, ale większe aplikacje będą w stanie zidentyfikować powtarzające się stany — na przykład adres URL poprzednio odwiedzanej strony internetowej. W rzeczywistości

użyjemy takich technik, aby uniknąć cykli i powtórzeń w późniejszych przykładach wymienionych w następnej sekcji.

Niektóre programy mogą również wymagać rejestracji pełnych ścieżek dla każdego obserwowanego stanu, aby mogły zgłaszać rozwiązania po zakończeniu. W takich przypadkach każdy element na stosie lub w kolejce schematu nierekursywnego może być pełną listą ścieżek, która wystarcza do zapisu odwiedzanych stanów i zawiera następny element do zbadania na obu końcach.

Zauważ też, że standardowy Python ogranicza głębokość stosu wywołań środowiska wykonawczego — kluczowego komponentu dla programów korzystających z wywołań rekurencyjnych — w celu wychwycenia błędów nieskończonej rekurencji. Aby zwiększyć domyślny rozmiar stosu, możemy użyć modułu sys:

```
>>> sys.getrecursionlimit()                      # Domyślny limit pozwala na 1000
wywołań rekurencyjnych

1000

>>> sys.setrecursionlimit(10000)                  # Zwiększamy limit zagnieżdżeń
>>> help(sys.setrecursionlimit)                  # Wyświetlamy więcej informacji na
ten temat
```

Maksymalna wartość tego parametru może się różnić w zależności od platformy. Nie jest to wymagane w przypadku programów, które używają stosów lub kolejek, aby uniknąć wywołań rekurencyjnych i uzyskania większej kontroli nad procesem trajektorii.

Więcej przykładów rekurencji

Choć przykład, który zaprezentujemy w tym podrozdziale, jest raczej akademicki, to jednak jest reprezentatywny dla większej klasy programów; drzewa dziedziczenia i łańcuchy importu modułów mogą na przykład wykazywać podobnie ogólne struktury, a struktury obliczeniowe, takie jak permutacje, mogą wymagać dowolnie wielu zagnieżdżonych pętli. W rzeczywistości użyjemy rekurencji w takich rolach w bardziej realistycznych przykładach w dalszej części tej książki:

- w rozdziale 20. moduł *permute.py* wykorzystuje rekurencję do arbitralnego mieszania sekwencji;
- w rozdziale 25. moduł *reloadall.py* wykorzystuje rekurencję do śledzenia łańcuchów importów;
- w rozdziale 29. moduł *classtree.py* śledzi drzewa dziedziczenia klas;
- w rozdziale 31. moduł *lister.py* również śledzi drzewa dziedziczenia;
- w dodatku D rekurencję wykorzystują rozwiązania dwóch ćwiczeń na końcu tej części książki: zliczanie i silnia.

Drugi i trzeci moduł potrafi wykrywać stany odwiedzone, aby uniknąć cykli i powtórzeń.

Choć ze względu na prostotę i wydajność kodu proste pętle powinny mieć przewagę nad rekurencją dla iteracji liniowych, przekonasz się, że rekurencja jest niezbędną techniką w scenariuszach, które pokażemy w późniejszych przykładach.

Co więcej, należy być świadomym potencjalnych konsekwencji *niezamierzonych* rekurencji w programach. Jak się przekonamy w dalszej części książki, również własne implementacje specjalnych metod klas, jak `__setattr__` i `__getattribute__`, a nawet `__repr__`, bywają podatne na rekurencję, jeżeli zostaną użyte w sposób niewłaściwy. Rekurencja to potężne narzędzie, ale działa ona najlepiej wówczas, gdy jest dobrze rozumiana i kontrolowana.

Obiekty funkcji – atrybuty i adnotacje

Funkcje w Pythonie są bardziej elastyczne, niż można by się spodziewać. Jak mieliśmy już okazję się przekonać, funkcje w Pythonie są czymś więcej niż jedynie specyfikacją operacji dla kompilatora: funkcje są pełnoprawnymi *obiektami*, zapisanymi w dedykowanych obszarach pamięci. Dzięki temu można je przekazywać w programach i wywoływać niebezpośrednio. Można również wykonywać na nich operacje, które są rzadko kojarzone z funkcjami: przypisywać atrybuty i adnotacje.

Pośrednie wywołania funkcji – obiekty „pierwszej klasy”

Ponieważ funkcje w Pythonie są obiektami, można pisać programy, które wykorzystują je w sposób generyczny. Obiekty funkcji można przypisywać do zmiennych, przekazywać do innych funkcji jako parametry, osadzać w strukturach danych, zwracać w wyniku działania innych funkcji i wykonywać na nich wiele innych operacji, analogicznie jak to się dzieje z liczbami czy ciągami znaków. Obiekty funkcji obsługują specjalne operacje: można je wywoływać, przekazując w nawiasach parametry wywołania w wyrażeniu funkcji. Mimo tych cech szczególnych funkcje należą do tej samej kategorii co pozostałe obiekty.

Jest to zwykle nazywane *modelem obiektowym pierwszej klasy* (ang. *first-class object model; model FCO*); jest on wszechobecny w Pythonie i jest niezbędnym elementem programowania funkcyjnego. Omówimy ten tryb programowania pełniej w tym i następnym rozdziale; ponieważ jego założenia opierają się na pojęciu funkcji, funkcje należy traktować jak dane.

We wcześniejszych przykładach mieliśmy już okazję zaobserwować przypadki generycznego użycia funkcji, ale warto zrobić małą powtórkę w celu wyraźnego podkreślenia faktu oparcia funkcji w Pythonie na modelu obiektowym. Na przykład nazwa użyta w instrukcji `def` nie jest przez Pythona traktowana w szczególny sposób. Jest to po prostu przypisanie nazwy w bieżącym zakresie traktowanym dokładnie tak samo jak w przypadku jego wykorzystania w wyrażeniu przypisania po lewej stronie znaku `=`. Po wywołaniu instrukcji `def` nazwa funkcji staje się referencją do obiektu funkcji: ten sam obiekt można przypisać do innej nazwy i wywołać tę samą funkcję przez referencję:

```
>>> def echo(message):                      # Nazwa echo przypisana do obiektu
    funkcji
    print(message)
>>> echo('Wywołanie bezpośrednie')        # Wywołanie obiektu przez oryginalną
nawę
Direct call
>>> x = echo                                # Teraz również zmienna x zawiera
referencję do funkcji
>>> x('Wywołanie pośrednie!')              # Wywołanie obiektu przez nazwę
przez zastosowanie ()
Wywołanie pośrednie!
```

Argumenty są przekazywane przez przypisanie obiektów, zatem funkcje można przekazywać w argumentach innych funkcji. Funkcja, której przekazano inną funkcję, może ją wywołać, dodając nawiasy do nazwy i przekazując w nich argumenty wywoływanej funkcji:

```
>>> def indirect(func, arg):
```

```

        func(arg)                      # Wywołanie przekazanego obiektu
przez zastosowanie ()

>>> indirect(echo, 'Wywołanie argumentu!') # Przekazanie funkcji do innej
funkcji

Wywołanie argumentu!

```

Obiektów funkcji można użyć również w charakterze elementów innych struktur danych, tak samo jak by to były liczby czy ciągi znaków. W poniższym przykładzie funkcję zapisujemy w liście krotek, tworząc coś na kształt tabeli operacji. W Pythonie tego typu struktury danych mogą zawierać dowolne typy obiektów, zatem i w tym przypadku nie dzieje się nic szczególnego:

```

>>> schedule = [(echo, 'Mielonka!'), (echo, 'Szynka!')]
>>> for (func, arg) in schedule:
        func(arg)                      # Wywołanie funkcji osadzonych w
kontenerze

Mielonka!
Szynka!

```

Powyższy kod po prostu przechodzi przez kolejne elementy listy `schedule`, wywołując funkcję `echo` z argumentem, będącym drugim elementem krotki (w deklaracji pętli następuje rozpakowanie krotki; informacje o tej technice można znaleźć w rozdziale 13.). Jak mieliśmy okazję zaobserwować w przykładach z rozdziału 17., funkcje można tworzyć w innych funkcjach i zwracać je w celu wywołania w innym miejscu programu — domknięcie utworzone w tym trybie zachowuje również stan z otaczającego zasięgu:

```

>>> def make(label):          # Utworzenie funkcji, bez wywołania
    def echo(message):
        print(label + ':' + message)
    return echo

>>> F = make('Mielonka')      # Zostaje zapisana etykieta przekazana do
funkcji w wewnętrznym zasięgu
>>> F('Szynka!')            # Wywołanie funkcji utworzonej w funkcji
make

Mielonka:Szynka!
>>> F('Jajka!')
Mielonka:Jajka!

```

Uniwersalny model obiektowy FCO i brak deklaracji typu sprawiają, że Python jest niezwykle elastycznym językiem programowania.

Introspekcja funkcji

Funkcje są obiektami, można więc przetwarzać je z użyciem standardowych narzędzi do obsługi obiektów. W rzeczywistości funkcje są znacznie bardziej elastyczne, niż można by się spodziewać. Na przykład po utworzeniu funkcji możemy ją po prostu wywołać, tak jak robiliśmy to do tej pory:

```
>>> def func(a):
```

```

b = 'mielonka'

return b * a

>>> func(5)
'mielonkamielonkamielonkamielonkamielonka'

```

Jednak wyrażenie wywołania to tylko jedna z operacji dostępnych obiektom funkcji. Możemy również dokonać inspekcji atrybutów, wykorzystując standardowe narzędzia (poniżej przykład wywołany w 3.3, ale wyniki dla 2.x są zbliżone):

```

>>> func.__name__
'func'

>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
... kolejne elementy zostały celowo pominięte (razem 34)...
['__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']

```

Narzędzia introspekcji pozwalają również na eksplorację szczegółów implementacyjnych: funkcje posiadają *obiekty kodu*, które zawierają takie szczegółы dotyczące implementacji, jak zmienne lokalne czy zadeklarowane argumenty.

```

>>> func.__code__
<code object func at 0x00000000021A6030, file "<stdin>", line 1>
>>> dir(func.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__',
... kolejne elementy zostały celowo pominięte (razem 37)...
['co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
 'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount',
 'co_lnotab',
 'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']
>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1

```

Twórcy narzędzi mogą wykorzystać takie informacje do zarządzania funkcjami (w rozdziale 39. my również będziemy mieli okazję skorzystać z tych możliwości podczas implementowania mechanizmu walidacji argumentów funkcji w dekoratorach).

Atrybuty funkcji

Obiekty funkcji nie są ograniczone do zdefiniowanych w systemie atrybutów wykorzystanych w powyższym punkcie. Jak mieliśmy okazję zobaczyć w rozdziale 17., istnieje możliwość

definiowania własnych atrybutów funkcji (dostępna od wersji 2.1 Pythona).

```
>>> func
<function func at 0x000000000296A1E0>
>>> func.count = 0
>>> func.count += 1
>>> func.count
1
>>> func.handles = 'Button-Press'
>>> func.handles
'Button-Press'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
... i wiele więcej; w wersji 3.x wszystkie pozostałe nazwy mają podwójne
podkreślenia, dzięki czemu nie będą kolidowały z Twoimi nazwami...
__str__, '__subclasshook__', 'count', 'handles']
```

Wewnętrzne dane implementacyjne przechowywane w funkcjach są zgodne z konwencjami nazewnictwa zapobiegającymi konfliktom z bardziej dowolnymi nazwami atrybutów, które możesz przypisywać samodzielnie. W wersji 3.x nazwy wszystkich funkcji wewnętrznych rozpoczynają się i kończą podwójnymi znakami podkreślenia („__X__”); wersja 2.x korzysta z tego samego schematu, ale przypisuje także niektóre nazwy rozpoczynające się od „func_X”:

```
c:\code> py -3
>>> def f(): pass
>>> dir(f)
...uruchom samodzielnie, aby zobaczyć wynik...
>>> len(dir(f))
34
>>> [x for x in dir(f) if not x.startswith('__')]
[]

c:\code> py -2
>>> def f(): pass
>>> dir(f)
...uruchom samodzielnie, aby zobaczyć wynik...
>>> len(dir(f))
31
>>> [x for x in dir(f) if not x.startswith('__')]
['func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc',
```

```
'func_globals', 'func_name']
```

Jeżeli starasz się nie nazywać atrybutów w ten sam sposób, możesz bezpiecznie korzystać z przestrzeni nazw funkcji, tak jakby to była Twoja własna przestrzeń nazw lub zasięg.

Jak widzieliśmy w tym rozdziale, takich atrybutów można użyć do bezpośredniego dołączenia *informacji o stanie* do obiektów funkcji; nie trzeba więc używać innych technik, takich jak zmienne globalne, zmienne nielokalne czy klasy. W przeciwieństwie do obiektów innych niż lokalne takie atrybuty są dostępne w dowolnym miejscu samej funkcji, nawet spoza jej kodu.

W pewnym sensie jest to sposób implementacji odpowiednika mechanizmu „statycznych zmiennych lokalnych” dostępnego w innych językach programowania. Są to zmienne, których nazwy są lokalne dla funkcji, ale ich wartość jest zachowana po zakończeniu jej działania. Atrybuty są połączone z obiektami, nie zasięgami nazw (i należy się do nich odwoływać poprzez nazwę funkcji w kodzie), ale efekt końcowy jest podobny.

Co więcej, jak dowiedzieliśmy się w rozdziale 17., kiedy atrybuty są dołączone do funkcji generowanych przez inne *funkcje fabrykujące*, obsługują one także przechowywanie informacji o stanie dla każdego wywołania, podobnie jak nielokalne domknięcia i atrybuty instancji klasy.

Adnotacje funkcji w Pythonie 3.x

W Pythonie 3.x (ale nie w 2.x) istnieje możliwość dołączania *dodatkowych informacji* do obiektu funkcji. Są to dowolne, zdefiniowane przez użytkownika informacje dotyczące argumentów przekazywanych do funkcji i wyników z niej zwracanych. Python oferuje specjalną składnię do definiowania adnotacji, ale nie wykorzystuje ich w jakikolwiek sposób. Adnotacje funkcji są mechanizmem zupełnie opcjonalnym, a gdy są zdefiniowane, zostają zapisane w atrubucie `__annotations__` obiektu funkcji i mogą być użyte przez inne narzędzia, które na przykład mogą wykorzystywać adnotacje w kontekście testowania błędów.

Specyficzne dla Pythona 3.x argumenty będące słowami kluczowymi poznaliśmy w poprzednim rozdziale; adnotacje stanowią kolejny mechanizm jeszcze bardziej uogólniający składnię nagłówka funkcji. Weźmy na przykład następującą funkcję przyjmującą trzy argumenty i zwracającą ich sumę:

```
>>> def func(a, b, c):
    ...
    return a + b + c
>>> func(1, 2, 3)
6
```

Z punktu widzenia składni adnotacje funkcji są zapisane w wierszach nagłówka funkcji jako wyrażenia związane z argumentami i zwracanymi wartościami. W przypadku argumentów adnotacja występuje po dwukropku następującym po nazwie atrybutu, w przypadku wartości zwracanych do adnotacji służy sekwencja znaków `->` następująca po liście argumentów. Poniższy przykład przedstawia wersję tej samej funkcji z adnotacjami wszystkich trzech argumentów i wyniku:

```
>>> def func(a: 'mielonka', b: (1, 10), c: float) -> int:
    ...
    return a + b + c
...
>>> func(1, 2, 3)
6
```

Wywołanie funkcji zawierającej adnotacje odbywa się tak samo jak każdej innej funkcji — w przypadku zdefiniowania adnotacji Python po prostu zapisuje je w słowniku i dołącza do obiektu funkcji. Nazwy argumentów stają się kluczami, adnotacja wartości zwracanej jest przechowywana pod kluczem `return`, o ile istnieje (co wystarcza, ponieważ tego zarezerwowanego słowa nie można użyć jako nazwy argumentu), a wartości kluczy adnotacji są przypisywane do wyników wyrażeń adnotacji:

```
>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'mielonka', 'return': <class
'int'>}
```

Adnotacje są obiektami Pythona dołączonymi do obiektu Pythona, dzięki czemu ich obsługa jest bardzo prosta. W poniższym przykładzie definiujemy adnotacje dla dwóch z trzech argumentów funkcji, po czym przeglądamy wszystkie adnotacje, wykorzystując standardowe narzędzia języka:

```
>>> def func(a: 'mielonka', b, c: 99):
    return a + b + c
>>> func(1, 2, 3)
6
>>> func.__annotations__
{'c': 99, 'a': 'mielonka'}
>>> for arg in func.__annotations__:
    print(arg, '=>', func.__annotations__[arg])
c => 99
a => mielonka
```

Warto tu zwrócić uwagę na dwa ciekawe szczegóły. Po pierwsze, korzystając z adnotacji, nie musimy rezygnować z definiowania *domyślnych* wartości argumentów: adnotacja (wraz ze znakiem `:`) występuje przed deklaracją wartości domyślnej (wraz z jej znakiem `=`). W poniższym przykładzie zapis `a: 'mielonka' = 4` oznacza, że wartością domyślną argumentu `a` jest 4, a argument posiada adnotację `'mielonka'`.

```
>>> def func(a: 'mielonka' = 4, b: (1, 10) = 5, c: float = 6) -> int:
    return a + b + c
>>> func(1, 2, 3)
6
>>> func()                      # 4 + 5 + 6      (wartości domyślne)
15
>>> func(1, c=10)                # 1 + 5 + 10    (parametry kluczowe działają
standardowo)
16
>>> func.__annotations__
{'c': <class 'float'>, 'b': (1, 10), 'a': 'mielonka', 'return': <class
'int'>}
```

Po drugie warto zauważyc, że *białe znaki* są opcjonalne: pomiędzy elementami w nagłówku funkcji mogą wystąpić białe znaki, ale nie muszą, jednak pominięcie ich może dla niektórych użytkowników zmniejszyć czytelność kodu (i jak znam życie, znacząco zwiększyć czytelność kodu dla innych):

```
>>> def func(a:'mielonka'=4, b:(1,10)=5, c:float=6)->int:  
    return a + b + c  
  
>>> func(1, 2)                      # 1 + 2 + 6  
9  
  
>>> func.__annotations__  
{'c': <class 'float'>, 'b': (1, 10), 'a': 'mielonka', 'return': <class 'int'>}
```

Adnotacje funkcji są nowym elementem języka i zostały wprowadzone w wersji 3.x. Wiele z ich potencjalnych zastosowań nadal pozostaje nieodkrytych. Łatwo wyobrazić sobie użycie adnotacji do określania ograniczeń typów lub wartości — bardziej rozbudowane API mogą wykorzystać tę cechę języka do rejestracji informacji o funkcji.

W rozdziale 39. spróbujemy wykorzystać tę możliwość jako alternatywę dla *argumentów dekoratorów funkcji* (jest to bardziej ogólna koncepcja, w której informacja jest zakodowana poza nagłówkiem funkcji, a więc nie jest ograniczona do jednej roli). Podobnie jak sam Python, adnotacje są narzędziem, którego zastosowanie jest ograniczone jedynie wyobraźnią programisty.

Na koniec należy zaznaczyć, że adnotacje działają wyłącznie w instrukcji `def`, nie są dostępne w wyrażeniach `lambda`, ponieważ składnia `lambda` sama w sobie narzuca ograniczenia definiowanym funkcjom. I w ten sposób przechodzimy do następnego tematu.

Funkcje anonimowe — lambda

Obok instrukcji `def` Python udostępnia również formę wyrażenia generującego obiekty funkcji. Ze względu na podobieństwo do narzędzia z języka LISP nosi ono nazwę `lambda`^[1]. Tak jak `def`, wyrażenie to tworzy funkcję, którą można wywołać później, jednak zwraca tę funkcję, zamiast przypisywać ją do nazwy. Z tego powodu wyrażenia `lambda` nazywane są czasami funkcjami *anonimowymi* (nienazwanymi). W praktyce często wykorzystywane są jako sposób skrótowego zapisania definicji funkcji lub opóźnienia wykonania fragmentu kodu.

Podstawy wyrażeń lambda

Na ogólną formę wyrażenia `lambda` składa się słowo kluczowe `lambda`, po którym następuje jeden lub większa liczba argumentów (dokładnie tak samo, jak argumenty umieszczone w nawiasach w nagłówku instrukcji `def`), a po nich, po dwukropku, wyrażenie.

```
lambda argument1, argument2, ... argumentN : wyrażenie wykorzystujące  
argumenty
```

Obiekty funkcji zwarcane przez wykonanie wyrażeń `lambda` działają dokładnie tak samo jak te utworzone i przypisane przez instrukcję `def`. Istnieje jednak kilka różnic sprawiających, że wyrażenie `lambda` staje się użyteczne w pewnych wyspecjalizowanych rolach.

- **lambda jest wyrażeniem, a nie instrukcją.** Z tego powodu może się pojawiać w miejscach, w których użycie def w składni Pythona nie będzie dozwolone — wewnątrz literała listy czy w wywołaniu funkcji. W bloku def można odwoływać się do funkcji według nazwy, ale same funkcje muszą być tworzone gdzie indziej. Jako wyrażenie lambda zwraca również wartość (nową funkcję), do której opcjonalnie można przypisać nazwę. W przeciwnieństwie do tego instrukcja def zawsze przypisuje nową funkcję do nazwy z nagłówka, zamiast zwracać ją jako wynik.
- **Ciałem funkcji lambda jest pojedyncze wyrażenie, a nie blok instrukcji.** Ciało wyrażenia lambda jest podobne do tego, co umieszcza się w instrukcji return ciała instrukcji def. Wynik wypisuje się po prostu jako gołe wyrażenie, zamiast w jawnym sposobie zwracać. Ponieważ lambda ograniczona jest do wyrażenia, jest mniej uniwersalna od def — ilość logiki, jaką można wstawić do ciała tego wyrażenia bez używania instrukcji, takich jak if, jest stosunkowo ograniczona. Taki zabieg jest celowy — ogranicza on zagnieżdżanie programu: wyrażenie lambda zostało zaprojektowane z myślą o zapisywaniu prostych funkcji, natomiast instrukcja def zajmuje się większymi zadaniami programistycznymi.

Poza tymi różnicami def i lambda wykonują ten sam rodzaj zadań. Widzieliśmy na przykład, w jaki sposób tworzy się funkcję za pomocą instrukcji def:

```
>>> def func(x, y, z): return x + y + z
>>> func(2, 3, 4)
9
```

Ten sam efekt można uzyskać za pomocą wyrażenia lambda, w jawnym sposobie przypisując jego wynik do zmiennej, za pomocą której można później wywołać funkcję:

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

W powyższym kodzie zmienna f przypisywana jest do obiektu funkcji tworzonego przez wyrażenie lambda. W ten sam sposób działa instrukcja def, jednak przypisanie jest tam automatyczne.

Domyślne wartości argumentów działają w wyrażeniach lambda w taki sam sposób jak w instrukcjach def.

```
>>> x = (lambda a="raz", b="dwa", c="trzy": a + b + c)
>>> x("las")
'lasdwatrzy'
```

Kod w ciele wyrażenia lambda przestrzega tych samych reguł przeszukiwania zakresów co kod wewnątrz instrukcji def. Wyrażenia lambda wprowadzają zasięg lokalny — dokładnie tak, jak zagnieżdżone instrukcje def — który automatycznie (dzięki regule LEGB i temu, o czym mówiliśmy w rozdziale 17.) widzi nazwy w funkcjach je zawierających, module oraz zasięgu wbudowanym.

```
>>> def knights( ):
        title = 'Sir'
        action = (lambda x: title + ' ' + x) # Tytuł w zasięgu instrukcji def
        zawierającej wyrażenie
```

```

# lambda
    return action                                # Zwrócenie obiektu funkcji

>>> msg = act('robin')                         # 'robin' przekazane do x
>>> msg
'Sir robin'
>>> act                                         # act: funkcja, a nie jej
rezultat
<function knights.<locals>.<lambda> at 0x00000000029CA488>

```

W powyższym przykładzie w wersjach Pythona starszych od 2.2 wartość zmiennej `title` była zamiast tego przekazywana jako domyślna wartość argumentu. Jeżeli nie pamiętasz, dlaczego tak było, możesz to zawsze sprawdzić, wracając do opisu zasięgów z rozdziału 17.

Po co używamy wyrażeń lambda

Ogólnie rzecz biorąc, wyrażenia `lambda` przydają się jako rodzaj skrótu funkcji, który pozwala osadzać definicję funkcji w kodzie ją wykorzystującym. Są one całkowicie opcjonalne i zamiast nich zawsze możesz użyć `def`; co więcej, *powinieneś* tak zrobić, jeżeli Twoja funkcja wymaga mocy pełnych instrukcji, których wyrażenie `lambda` nie jest w stanie łatwo dostarczyć. Wyrażenia `lambda` na ogół są prostymi konstrukcjami, przydatnymi w sytuacjach, w których musimy osadzić niewielkie fragmenty kodu wykonywalnego.

Na przykład niebawem przekonasz się, że programy obsługujące wywołań zwrotnych są często zapisywane jako wyrażenia `lambda` osadzone bezpośrednio w liście argumentów wywołania rejestrującego, zamiast być zdefiniowane za pomocą instrukcji `def` w innym miejscu pliku (przykład takiego działania znajduje się w ramce „Warto pamiętać: wywołania zwrotne z użyciem wyrażeń `lambda`” w dalszej części tego rozdziału).

Wyrażenia `lambda` są również często wykorzystywane do tworzenia *tablic skoków* (ang. *jump table*), będących listami lub słownikami działań, które mają być wykonane na żądanie. Na przykład:

```

L = [(lambda x: x**2),                               # Wewnętrzna definicja funkcji
      (lambda x: x**3),
      (lambda x: x**4)]                                # Lista trzech wywoływanego funkcji

for f in L:
    print f(2)                                      # Wyświetla 4, 8, 16
    print f[0](3)                                    # Wyświetla 9

```

Wyrażenie `lambda` jest najbardziej przydatne jako skrót dla `def`, kiedy musimy zmieścić niewielkie fragmenty kodu wykonywalnego w miejscach, w których instrukcje będą niepoprawne z punktu widzenia składni. Poniższy fragment kodu tworzy na przykład listę trzech funkcji, osadzając wyrażenia `lambda` wewnętrz literała listy. Instrukcja `def` nie może się pojawić wewnątrz literała listy, ponieważ jest instrukcją, a nie wyrażeniem. Odpowiednik w postaci tradycyjnej instrukcji `def` wymagałby zastosowania tymczasowych nazw funkcji definiowanych poza kontekstem ich użycia.

```
def f1(x): return x ** 2
```

```

def f2(x): return x ** 3           # Definiujemy nazwane funkcje
def f3(x): return x ** 4
L = [f1, f2, f3]                  # Odwołanie przez nazwę
for f in L:
    print(f(2))                  # Wyświetla 4, 8, 16
print(L[0](3))                   # Wyświetla 9

```

Wielotorowe rozgałęzienia kodu — finał

W rzeczywistości to samo można zrobić za pomocą słowników czy innych struktur danych, które służą w Pythonie do budowania tabeli działań. Oto kolejny przykład do wykonania w sesji interaktywnej:

```

>>> key = 'już'
>>> {'mam': (lambda: 2 + 2),
      'już': (lambda: 2 * 4),
      'jeden': (lambda: 2 ** 6)}[key]()
8

```

W powyższym kodzie, kiedy Python tworzy tymczasowy słownik, każde z zagnieżdżonych wyrażeń `lambda` generuje i pozostawia funkcję, którą można wywołać później. Indeksowanie po kluczu powoduje pobranie jednej z funkcji, a nawiasy wymuszają jej wywołanie. W takiej postaci słownik staje się bardziej uniwersalnym narzędziem rozgałęziania kodu od tego, co mogłem pokazać w omówieniu instrukcji `if` z rozdziału 12.

Aby uzyskać to samo bez użycia wyrażeń `lambda`, musielibyśmy utworzyć gdzieś w pliku, poza słownikiem, w którym funkcje mają być użyte, trzy instrukcje `def` oraz odwoływać się do tej funkcji po nazwie.

```

>>> def f1(): return 2 + 2
>>> def f2(): return 2 * 4
>>> def f3(): return 2 ** 6
>>> key = 'jeden'
>>> {'mam': f1, 'już': f2, 'jeden': f3}[key]()
64

```

Takie rozwiązanie również działa, jednak instrukcje `def` mogą być w pliku mocno od siebie oddalone, nawet jeśli składają się jedynie z niewielkiego fragmentu kodu. *Bliskość kodu* udostępniana przez wyrażenia `lambda` okazuje się szczególnie wygodna w przypadku funkcji, które będą używane w tylko jednym kontekście. Jeżeli trzy wymienione tu funkcje nie przydadzą się nam nigdzie indziej, osadzenie ich definicji wewnątrz słownika w postaci wyrażeń `lambda` ma sens. Co więcej, forma instrukcji wymaga nadania tym trzem niewielkim funkcjom nazw, które potencjalnie mogą pozostać w konflikcie z innymi nazwami z tego pliku (niezbyt prawdopodobne, ale niemniej jednak możliwe)[\[2\]](#).

Wyrażenia `lambda` przydają się również w listach argumentów funkcji, gdzie pozwalają na umieszczenie tymczasowych definicji funkcji nieużywanych nigdzie indziej w naszym programie. Przykłady takiego zastosowania zobaczymy nieco później, przy okazji omawiania funkcji `map`.

Jak (nie) zaciemniać kodu napisanego w Pythonie

Fakt, że ciało funkcji `lambda` musi być pojedynczym wyrażeniem (a nie serią instrukcji), wydaje się nakładać na nas ograniczenia w zakresie tego, ile logiki można upakować w jednym `lambda`. Jeżeli jednak wiemy, co robimy, większość instrukcji możemy w Pythonie zapisać za pomocą ich odpowiedników będących wyrażeniami.

Na przykład, jeżeli chcemy wyświetlić coś z poziomu ciała funkcji `lambda`, to w Pythonie 3.x wystarczy użyć `print(x)`, które będzie wyrażeniem wywołania, zamiast instrukcji, a w Pythonie 2.x lub 3.x użyć wyrażenia `sys.stdout.write(str(x) + '\n')`, aby mieć pewność, że jest to wyrażenie przenośne (jak zapewne pamiętasz z rozdziału 11., w taki właśnie sposób działa instrukcja `print`). I podobnie, aby osadzić logikę w wyrażeniu `lambda`, można użyć trójelementowego wyrażenia `if/else` przedstawionego w rozdziale 12. lub opisanego w tym samym rozdziale (i nieco bardziej podchwytnego) odpowiednika — kombinacji `and/or`. Jak pamiętasz, poniższą instrukcję:

```
if a:  
    b  
else:  
    c
```

można zastąpić jednym z odpowiadających jej wyrażeń:

```
b if a else c  
((a and b) or c)
```

Ponieważ takie wyrażenia można umieszczać wewnętrz wyrażeń `lambda`, mogą one zostać wykorzystane do implementowania logiki wyboru wewnętrz funkcji `lambda`.

```
>>> lower = (lambda x, y: x if x < y else y)  
>>> lower('bb', 'aa')  
'aa'  
>>> lower('aa', 'bb')  
'aa'
```

Co więcej, jeżeli potrzebujemy wewnętrz wyrażenia `lambda` wykonać pętlę, możemy również osadzić w nim elementy takie, jak wywołania `map` czy wyrażenia list składanych (narzędzia omówione we wcześniejszych rozdziałach, do których powrócimy w dalszej części niniejszego rozdziału).

```
>>> import sys  
>>> showall = (lambda x: map(sys.stdout.write, x))      # w wersji 3.x musisz  
użyć listy  
>>> t = showall(['mielonka\n', 'tost\n', 'jajka\n'])      # w wersji 3.x możesz  
użyć print  
mielonka  
tost  
jajka  
>>> showall = lambda x: [sys.stdout.write(line) for line in x]  
>>> t = showall(['patrz\n', 'na\n', 'życie\n', 'z\n', 'humorem\n'])
```

```
patrz
na
życie
z
humorem

>>> showall = lambda x: [print(line, end='') for line in x]      # to samo,
tylko dla wersji 3.x

>>> showall = lambda x: print(*x, sep='', end='')                  # to samo,
tylko dla wersji 3.x
```

Emulacja instrukcji za pomocą wyrażeń posiada pewne ograniczenia: na przykład nie można bezpośrednio osiągnąć efektu instrukcji przypisania, chociaż narzędzi takie jak wbudowana funkcja `setattr`, atrybut `_dict_` przestrzeni nazw i metody zmieniające obiekty mutowalne w miejscu mogą spełniać podobną rolę, a techniki programowania funkcyjnego mogą zaprowadzić Cię w głęb mrocznej sfery kodu.

Skoro już zaprezentowałem te sztuczki, czuję się zobowiązany poprosić o stosowanie ich tylko w ostateczności. Bez zachowania ostrożności łatwo mogą się one przyczyniać do powstawania mało czytelnego (inaczej mówiąc, *zaciemnionego*) kodu w Pythonie. Jak zawsze proste jest lepsze od złożonego, a jawne lepsze od niejawnego. Z tego powodu pełne instrukcje niemal zawsze będą lepsze od zawiłych, skomplikowanych wyrażeń. Z drugiej strony, takie techniki mogą się nam czasem przydać, o ile używane będą z rozsądkiem.

Zasięgi: wyrażenia lambda również można zagnieźdzać

Wyrażenia `lambda` najbardziej zyskały na wprowadzeniu wyszukiwania w zasięgach funkcji zagnieżdżonych (czyli *E* z reguły LEGB omówionej w rozdziale 17.). Na przykład w poniższym kodzie wyrażenie `lambda` pojawia się wewnątrz instrukcji `def` (typowy przypadek), dlatego możemy uzyskać dostęp do wartości, jaką zmienna `x` miała w zasięgu funkcji ją zawierającej w momencie wywołania tej funkcji.

```
>>> def action(x):
    return (lambda y: x + y)                      # Utworzenie i zwrócenie funkcji,
zapamiętanie x

>>> act = action(99)

>>> act
<function action.<locals>.<lambda> at 0x00000000029CA2F0>

>>> act(2)                                      # Wywołanie wyniku funkcji action
101
```

Czego nie było widać w omówieniu zakresów funkcji zagnieżdżonych w poprzednim rozdziale, to to, że wyrażenie `lambda` ma dostęp do zmiennych z zawierającego je innego wyrażenia `lambda`. Taki przypadek jest nieco nietypowy, ale wyobraźmy sobie sytuację, w której instrukcję `def` z poprzedniego przykładu zastępujemy wyrażeniem `lambda`.

```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
```

```
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

W powyższym kodzie zagnieźdzona struktura lambda tworzy funkcję, która przy wywołaniu tworzy kolejną funkcję. W obu przypadkach kod zagnieżdzonego wyrażenia lambda ma dostęp do zmiennej `x` z zewnętrznego lambda. Takie rozwiązańe działa, choć jest nieco zagmatwane. W interesie czytelności kodu lepiej jest unikać zagnieżdżania wyrażeń lambda.

Warto pamiętać: wywołania zwrotne z użyciem wyrażeń lambda

Innym, bardzo często wykorzystywanym zastosowaniem wyrażeń lambda jest definiowanie funkcji zwrotnych dla API graficznego interfejsu użytkownika tkinter Pythona (ten moduł w wersji 2.x nosi nazwę Tkinter). Na przykład poniższy kod tworzy przycisk wyświetlający po jego naciśnięciu komunikat w konsoli (zakładając, że biblioteka tkinter jest dostępna na Twoim komputerze; jest domyślnie instalowana w systemach Windows, macOS, Linux i innych).

```
import sys
from tkinter import Button, mainloop # Tkinter w 2.x
x = Button(
    text='Naciśnij mnie',
    command=(lambda:sys.stdout.write('Mielonka\n'))) # w wersji 3.x:
print()
x.pack() # opcjonalne w trybie konsoli
mainloop()
```

W tym przykładzie program obsługuje wywołania zwrotnego rejestrowany jest za pomocą przekazania funkcji wygenerowanej przy użyciu wyrażenia lambda do argumentu ze słowem kluczowym `command`. Przewaga lambda nad `def` polega tutaj na tym, że kod obsługujący naciśnięcie przycisku jest od razu na miejscu, osadzony w wywołaniu tworzącym sam przycisk.

W rezultacie lambda opóźnia wykonanie programu obsługi do momentu wystąpienia zdarzenia. Wywołanie `write` odbywa się w momencie naciśnięcia przycisku, a nie przy jego utworzeniu i w efekcie w momencie zaistnienia zdarzenia metoda `write „zna”` już ciąg znaków, który ma być wyświetlony.

Ponieważ reguły zakresów zagnieżdżonych odnoszą się również do wyrażeń lambda, od Pythona 2.2 są one łatwiejsze w użyciu w roli programów obsługi wywołań zwrotnych — automatycznie widzą zmienne z funkcji, w których są tworzone, i w większości przypadków nie wymagają już przekazywania wartości domyślnych. Jest to szczególnie przydatne w dostępie do specjalnego argumentu instancji `self`, który jest zmienną lokalną w funkcjach metod klasy zawierającej (więcej informacji na temat klas znajduje się w części VI książki).

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.display("mielonka")))
    def display(self, message):
```

...użycie komunikatu...

We wczesnych wersjach Pythona nawet obiekt `self` musiał być przekazywany do funkcji `lambda` z ustawieniami domyślnymi. Jak zobaczymy później, obiekty klasy z metodą `__call__` i powiązanymi metodami często występują również w rolach metod zwrotnych — więcej szczegółowych informacji na ten temat znajdziesz w rozdziałach 30. i 31.

Narzędzia programowania funkcyjnego

Według większości definicji dzisiejszy Python łączy w sobie wiele paradygmatów programowania: proceduralny (ze swoimi podstawowymi instrukcjami), obiektowy (ze swoimi klasami) i funkcyjny. W przypadku tego ostatniego paradygmatu Python posiada zestaw wbudowanych komponentów używanych do *programowania funkcyjnego* — narzędzi, które używają funkcji do przetwarzania sekwencji i innych obiektów iterowalnych. Ten zestaw zawiera narzędzia, które wywołują funkcje na elementach iterowalnych (`map`); odfiltrowują elementy na podstawie funkcji testowej (`filter`), używają funkcji do przetwarzania par elementów i generowania wyników (`reduce`).

Chociaż granice są czasami nieco rozmyte, według większości definicjiarsenał programowania funkcyjnego Pythona obejmuje również model obiektowy FCO, który został omówiony już wcześniej, zagnieżdżone *domknięcia* zasięgów i anonimowe funkcje `lambda`, generatorы i elementy składane, którymi będziemy zajmować się w następnym rozdziale, a być może również *dekoratory* funkcji i klas prezentowane w końcowej części tej książki. Dla naszych potrzeb zakończmy bieżący rozdział krótkim przeglądem wbudowanych funkcji, które automatycznie używają innych funkcji do przetwarzania obiektów iterowalnych.

Odwzorowywanie funkcji na obiekty iterowalne – `map`

Jednym z najczęściej wykonywanych działań na listach i innych sekwencjach jest zastosowanie wybranej operacji do każdego ich elementu i zbieranie wyników — wybieranie kolumn w tabelach bazy danych, zwiększenie wysokości wynagrodzeń pracowników w firmie, analizowanie załączników wiadomości e-mail itd. Python posiada wiele narzędzi, które ułatwiają kodowanie takich operacji w całej kolekcji. Uaktualnienie wszystkich liczników na liście można na przykład z łatwością wykonać za pomocą pętli `for`.

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)                      # Dodanie 10 do każdego elementu
...
>>> updated
[11, 12, 13, 14]
```

Ponieważ jest to tak często wykonywana operacja, Python udostępnia odpowiednią funkcję wbudowaną, która wykonuje za nas większość pracy. Funkcja `map` służy do zastosowania

przekazanej funkcji na każdym elemencie sekwencji i zwraca listę zawierającą wszystkie wyniki jej wywołania, jak w poniższym przykładzie.

```
>>> def inc(x): return x + 10          # Funkcja do wykonania

>>> map(inc, counters)                # Zbieranie wyników
[11, 12, 13, 14]
```

Funkcję `map` pokazywaliśmy pokrótko w rozdziałach 13. i 14. jako sposób na zastosowanie wbudowanej funkcji do elementów obiektu iterowalnego. Tutaj wykorzystamy ją w nieco ciekawszy sposób, przekazując jej wybraną funkcję *zdefiniowaną przez użytkownika*, która należy zastosować do każdego elementu listy — `map` wywołuje funkcję `inc` na każdym elemencie listy i zbiera zwarcane wartości w listę. Pamiętaj, że w Pythonie 3.x funkcja `map` jest obiektem iterowalnym, zatem wyświetlenie wszystkich wyników jej działania wymaga wywołania funkcji `list`; nie jest to konieczne w wersji 2.x (jeżeli zapomniałeś o tym wymaganiu, powinieneś zatrzymać się na rozdziale 14.).

Ponieważ `map` oczekuje przekazania funkcji, jest kolejnym miejscem, w którym może pojawić się wyrażenie `lambda`.

```
>>> list(map(lambda x: x + 3), counters)      # Wyrażenie funkcji
[4, 5, 6, 7]
```

W powyższym kodzie funkcja dodaje 3 do każdego elementu z listy `counters`. Ponieważ funkcja ta nie będzie potrzebna nigdzie indziej, została zapisana za pomocą wyrażenia `lambda`. Takie zastosowanie `map` jest odpowiednikiem pętli `for`, dlatego z użyciem niewielkiej ilości dodatkowego kodu zawsze można samodzielnie utworzyć uniwersalne narzędzie odwzorowujące.

```
>>> def mymap(func, seq):
    res = []
    for x in seq: res.append(func(x))
    return res
```

Załóżmy, że funkcja `inc` ma taką postać, jak zaprezentowana ostatnio. Możemy wykonać za jej pomocą odwzorowanie na sekwencji, używając wbudowanej funkcji `map` lub naszego odpowiednika.

```
>>> list(map(inc, [1, 2, 3]))            # Wbudowana funkcja zwraca obiekt
iterowalny
[11, 12, 13]

>>> mymap(inc, [1, 2, 3])               # Nasza wersja zwraca listę
(patrz generatory)
[11, 12, 13]
```

Ponieważ jednak `map` jest funkcją wbudowaną, jest zawsze dostępna, zawsze działa w ten sam sposób i ma pewną przewagę w zakresie wydajności (jest zwykle szybsza od ręcznie napisanej pętli `for`, co udowodnimy w rozdziale 21.). Co więcej, funkcji `map` można używać na bardziej zaawansowane, niż dotychczas pokazano, sposoby. Przykładowo po podaniu kilku argumentów będących sekwencjami przesyła ona elementy pobrane z sekwencji równolegle jako osobne argumenty funkcji.

```
>>> pow(3, 4)                          # 3**4
```

```
>>> list(map(pow, [1, 2, 3], [2, 3, 4]))      # 1**2, 2**3, 3**4
[1, 8, 81]
```

W przypadku N sekwencji funkcja `map` oczekuje funkcji N -argumentowej. W powyższym kodzie funkcja `pow` w każdym wywołaniu przyjmuje dwa argumenty — po jednym z każdej sekwencji przekazanej do funkcji `map`. Zasymulowanie tego typu uogólnienia w powyższym kodzie nie wymaga wiele pracy, ale odłożę to na później, gdy poznamy narzędzie iteracyjne. Choć moglibyśmy symulować takie zachowanie, nie ma to sensu, skoro Python udostępnia szybką funkcję wbudowaną.

Wywołanie `map` jest podobne do wyrażeń list składanych omówionych w rozdziale 14., z którymi spotkamy się zresztą jeszcze w następnym rozdziale z nieco innej perspektywy:

```
>>> list(map(inc, [1, 2, 3, 4]))
[11, 12, 13, 14]
>>> [inc(x) for x in [1, 2, 3, 4]]                      # Użyj (), aby wygenerować
wyniki
[11, 12, 13, 14]
```

W niektórych przypadkach wywołanie funkcji `map` może być szybsze niż listy składane (np. podczas mapowania funkcji wbudowanej) i może wymagać mniejszej ilości kodu. Z drugiej strony, ponieważ funkcja `map` zamiast arbitralnie wybranego *wyrażenia* wywołuje określona *funkcję* dla każdego iterowanego elementu, jest mniej ogólnym narzędziem i często wymaga dodatkowych funkcji pomocniczych lub wyrażeń `lambda`. Co więcej, opakowywanie list składanych w nawiasach zwykłych zamiast w kwadratowych tworzy obiekt, który *generuje* wartości na żądanie w celu zaoszczędzenia pamięci i zwiększenia czasu reakcji, podobnie jak robi to funkcja `map` w wersji 3.x — jest to temat, którym zajmiemy się w następnym rozdziale.

Wybieranie elementów obiektów iterowalnych — funkcja `filter`

Funkcja `map` jest podstawowym i relatywnie prostym przedstawicielem klasy wbudowanych narzędzi Pythona wykorzystywanym w *programowaniu funkcyjnym*. Jej bliscy krewni, funkcje `filter` i `reduce`, wybierają elementy obiektu iterowalnego w oparciu o funkcję testującą (`filter`) lub stosują funkcję do par elementów (`reduce`).

Ponieważ funkcja `filter` (podobnie jak `reduce`) zwraca obiekt iterowalny, aby w wersji 3.x wyświetlić ich wyniki działania należy je przekształcić w listę. Poniższe wywołanie funkcji `filter` wybiera elementy sekwencji większe od zera.

```
>>> list(range(-5, 5))                                # Obiekt iterowalny w
wersji 3.x
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(filter(lambda x: x > 0, range(-5, 5)))    # Obiekt iterowalny w
wersji 3.x
[1, 2, 3, 4]
```

Z funkcją `filter` spotkaliśmy się krótko w ramce w rozdziale 12. oraz podczas omawiania zagadnień związanych z iteracjami w wersji 3.x (rozdział 14.). Pozycje w sekwencji lub obiekty iterowalne, dla których funkcja zwraca prawdziwy wynik, są dodawane do listy wyników.

Podobnie jak funkcja `map`, funkcja `filter` jest z grubsza odpowiednikiem pętli `for`, ale jest wbudowana, zwięzła i często szybka:

```
>>> res = []
>>> for x in range(-5, 5):
    if x > 0:
        res.append(x)

>>> res
[1, 2, 3, 4]
```

Również podobnie jak w przypadku funkcji `map`, funkcja `filter` może być emulowana przez listy składane z często lepszymi rezultatami (zwłaszcza gdy można uniknąć tworzenia nowej funkcji) lub za pomocą odpowiedniego *wyrażenia generatora*, gdy pożądane jest opóźnione wytwarzanie wyników, ale resztę tej historii opowiemy dopiero w kolejnym rozdziale:

```
>>> [x for x in range(-5, 5) if x > 0]          # Użyj (), aby
wygenerować wyniki
[1, 2, 3, 4]
```

Łączanie elementów obiektów iterowalnych – funkcja `reduce`

Funkcja `reduce`, która w wersji 2.x jest zwykłą funkcją wbudowaną, a w 3.x została przeniesiona do modułu `functools`, jest bardziej złożona. Argumentem wywołania tej funkcji może być obiekt iterowalny, ale sam obiekt funkcji nie jest iterowalny — zwraca pojedynczy wynik działania. Poniżej przedstawiamy przykłady dwóch wywołań funkcji `reduce`, które obliczają sumę oraz iloczyn elementów listy:

```
>>> from functools import reduce                      # Import tylko w
3.x, nie w 2.x
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

W każdym kroku funkcja `reduce` przekazuje bieżącą sumę lub iloczyn wraz z kolejnym elementem listy do przekazanej funkcji `lambda`. Domyślnie pierwszy element sekwencji inicjalizuje wartość początkową. Poniżej znajduje się odpowiednik pierwszego wywołania utworzony z wykorzystaniem pętli `for`, z dodawaniem zakodowanym na stałe wewnątrz pętli.

```
>>> L = [1, 2, 3, 4]
>>> res = L[0]
>>> for x in L[1:]:
    res = res + x

>>> res
```

Utworzenie własnej wersji funkcji `reduce` jest tak naprawdę dość proste. Poniższy listing przedstawia implementację emulującą większość jej możliwości, ujawniając zasadę działania.

```
>>> def myreduce(function, sequence):
    tally = sequence[0]
    for next in sequence[1:]:
        tally = function(tally, next)
    return tally

>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
15
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```

Funkcja wbudowana `reduce` pozwala na użycie opcjonalnego trzeciego argumentu, wstawianego przed elementami sekwencji, który służy jako wartość początkowa i domyślny wynik, gdy sekwencja jest pusta, ale odpowiednie rozbudowanie kodu naszej funkcji pozostawimy Ci do samodzielnego wykonania jako sugerowane ćwiczenie.

Jeżeli ta technika kodowania wzbudziła Twoje zainteresowanie, możesz również zwrócić uwagę na standardowy moduł `operator`, który udostępnia funkcje odpowiadające wbudowanym wyrażeniom, a więc jest przydatny w przypadku niektórych zastosowań narzędzi funkcyjnych (więcej informacji na ten temat możesz znaleźć w dokumentacji tego modułu Pythona):

```
>>> import operator, functools
>>> functools.reduce(operator.add, [2, 4, 6])                      # Dodawanie oparte na
funkcji
12
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Tak jak `map`, również funkcje `filter` oraz `reduce` obsługują zaawansowane techniki programowania funkcyjnego. Jak już wspominaliśmy, wielu użytkowników bardzo chętnie rozszerzyłoby zestaw narzędzi programowania funkcyjnego w Pythonie, tak aby obejmował również zagnieżdżone zasięgi (funkcje fabrykujące), anonimowe funkcje `lambda` (o których również mówiliśmy już wcześniej) oraz *generatory* i *wyrażenia składane*, do których powróćmy w kolejnym rozdziale.

Podsumowanie rozdziału

Niniejszy rozdział był omówieniem bardziej zaawansowanych zagadnień związanych z funkcjami — funkcji rekurencyjnych, adnotacji funkcji, wyrażeń funkcji `lambda`, narzędzi funkcyjnych, takich jak `map`, `filter` oraz `reduce`, a także ogólnych koncepcji związanych z projektowaniem funkcji. Powróciliśmy również do obiektów iterowalnych i list składanych, ponieważ są one powiązane z programowaniem funkcyjnym w takim samym stopniu jak z

instrukcjami pętli. Zanim jednak przejdziemy dalej, powinieneś upewnić się, że opanowałeś podstawy funkcji, wykonując quiz podsumowujący rozdział, a także ćwiczenia kończące tę część książki.

Sprawdź swoją wiedzę – quiz

1. W jaki sposób są ze sobą powiązane wyrażenia `lambda` oraz instrukcje `def`?
2. Jaki jest cel używania wyrażeń `lambda`?
3. Porównaj działanie funkcji `map`, `filter` i `reduce`.
4. Czym są adnotacje funkcji i w jaki sposób się ich używa?
5. Czym są funkcje rekurencyjne i w jaki sposób się ich używa?
6. Jakie są ogólne zalecenia dotyczące tworzenia funkcji?
7. Wymień trzy lub więcej sposobów, w jakie funkcje mogą przekazywać wyniki do obiektu wywołującego.

Sprawdź swoją wiedzę – odpowiedzi

1. Zarówno `lambda`, jak i `def` mogą tworzyć obiekty funkcji, które mogą być wywołane później. Ponieważ jednak `lambda` jest wyrażeniem, można ją wykorzystać do zagnieżdżenia definicji funkcji w miejscach, w których `def` nie jest dozwolone przez składnię. Wykorzystywanie wyrażenia `lambda` nigdy nie jest wymagane — zawsze można zamiast niego użyć instrukcji `def` i odnieść się do funkcji przez jej nazwę. Wyrażenia te przydają się jednak do osadzania niewielkich fragmentów opóźnionego kodu, z którego raczej nie będziemy korzystać w żadnym innym miejscu programu. Z punktu widzenia składni `lambda` zezwala na jedno wyrażenie zwracające wartość. Ponieważ nie obsługuje bloków instrukcji, nie nadaje się do implementowania większych funkcji.
2. Wyrażenia `lambda` pozwalają na definiowanie prostych fragmentów kodu wykonywalnego z opóźnieniem jego wykonania i możliwością zachowania stanu w postaci domyślnych argumentów i zmiennych zakresu zewnętrznego. Użycie `lambda` nigdy nie jest konieczne, zawsze istnieje możliwość zdefiniowania funkcji z użyciem instrukcji `def` i odwoływania się do niej po nazwie. Wyrażenia `lambda` są wygodne w sytuacjach, gdy istnieje potrzeba osadzenia kodu wykonywalnego z opóźnieniem jego wywołania, w przypadku gdy ten kod nie będzie użyty w innych miejscach aplikacji. Powszechnym zastosowaniem wyrażeń `lambda` jest kod definicji interfejsu GUI, gdzie stosuje się je z narzędziami programowania funkcyjnego, jak `map` i `filter`, które jako jednego z parametrów wymagają funkcji.
3. Te funkcje wbudowane służą do wykonywania funkcji na elementach sekwencji (obiektu iterowanego) i zwracania wyniku takiej operacji. Funkcja `map` po prostu przekazuje elementy sekwencji jako argumenty funkcji i zwraca sekwencję wyników takich wywołań, `filter` zwraca sekwencję złożoną z elementów, dla których podana funkcja zwraca wartość prawdziwą, `reduce` oblicza pojedynczą wartość, wykonując funkcję na akumulatorze i kolejnych elementach sekwencji. W przeciwnieństwie do

pozostałych dwóch funkcji funkcja `reduce` jest dostępna w module `functools` w wersji 3.x; w wersji 2.x jest to funkcja wbudowana.

4. Adnotacje funkcji, dostępne od wersji 3.x (3.0 i nowszych), są składniowymi ozdobami argumentów i wyników funkcji, które są gromadzone w słowniku przypisany do atrybutu `_annotations_` obiektu funkcji. Python nie przypisuje adnotacjom żadnego znaczenia semantycznego, ale po prostu zapisuje je do potencjalnego wykorzystania przez inne narzędzia.
5. Funkcje rekurencyjne bezpośrednio lub pośrednio wywołują same siebie w celu wykonania pętli operacji. Mogą być użyte do przetwarzania struktur danych o niezdefiniowanej strukturze, można ich również użyć jako ogólnego mechanizmu iteracyjnego (to zastosowanie jednak lepiej powierzyć pętlom, które zrealizują je szybciej i bardziej efektywnie z punktu widzenia pamięci). Rekurencję często można symulować lub zastępować kodem, który używa jawnych stosów lub kolejek, dających większą kontrolę nad przetwarzanymi elementami.
6. Funkcje powinny być niewielkimi, samodzielnymi fragmentami logiki aplikacji, posiadać prosty, uogólniony cel i komunikować się z innymi elementami aplikacji za pośrednictwem swoich argumentów wywołania i zwracanych wyników. Mogą również wykorzystywać mutowalne argumenty i za ich pośrednictwem przekazywać zmiany, ale preferowane są dwa pierwsze mechanizmy komunikacji funkcji z otoczeniem.
7. Funkcje mogą przekazywać wyniki swojego działania za pomocą instrukcji `return`, zmieniając przekazywane argumenty mutowalne i ustawiając zmienne globalne. Zmienne globalne na ogół nie powinny być stosowane do tego celu (z wyjątkiem bardzo szczególnych przypadków, takich jak programy wielowątkowe), ponieważ mogą utrudnić zrozumienie i poprawne użycie kodu. Instrukcja `return` zwykle sprawdza się najlepiej, ale modyfikowanie zmiennych mutowalnych również jest dopuszczalne (a nawet często przydatne), jeżeli wiesz, co robisz, i masz nad tym procesem pełną kontrolę. Funkcje mogą również wymieniać wyniki działania z urządzeniami systemowymi, takimi jak pliki i gniazda, ale omawianie tych zagadnień wykracza daleko poza zakres naszej książki.

[1] Nazwa `lambda` wydaje się odstraszać ludzi bardziej, niż powinna. Reakcja ta wydaje się wynikać z samej nazwy „`lambda`” — nazwy, która pochodzi z języka LISP, który z kolei zaczerpnął ją z rachunku lambda, będącego formą logiki symbolicznej. W Pythonie jest to jednak tylko słowo kluczowe wprowadzające odpowiednią składnię wyrażenia. Po odsunięciu na bok tego matematycznego dziedzictwa szybko przekonasz się, że funkcja `lambda` jest łatwiejsza w użyciu, niż myślisz: jest to po prostu alternatywny sposób kodowania funkcji, choć bez pełnych instrukcji, dekoratorów lub adnotacji używanych w wersji 3.x.

[2] Pewien student zauważył kiedyś, że można pominąć słownik tabeli skoków w takim kodzie, jeżeli nazwa funkcji jest taka sama jak klucz wyszukiwania łańcucha — aby uruchomić wywołanie, wystarczy wykonać polecenie `eval(funcname)()`. Chociaż w tym przypadku jest to prawdą, a czasem jest nawet bardzo przydatne, jak widzieliśmy wcześniej (np. w rozdziale 10.), użycie instrukcji `eval` jest względnie wolne (musi się skompilować i dopiero uruchamiać kod) i niezbyt bezpieczne (musisz mieć zaufanie do źródła, z którego pochodzi uruchamiany łańcuch znaków). Mówiąc bardziej ogólnie, w Pythonie tabele skoków są zazwyczaj obsługiwane przez metody polimorficzne: wywołanie metody wykonuje „właściwą operację” w zależności od typu obiektu. Odpowiedź na pytanie, dlaczego tak się dzieje, znajdziesz w części VI tej książki.

Rozdział 20. Listy składane i generatory

W tym rozdziale kontynuujemy omawianie zaawansowanych zagadnień związanych z funkcjami w kontekście zastosowania list składanych i narzędzi iteracyjnych wprowadzonych w rozdziale 4. i szerzej omówionych w rozdziale 14. Ponieważ *listy składane* są tak samo powiązane z funkcjami omawianymi w poprzednim rozdziale (takimi jak `map` czy `filter`) jak z pętlami `for`, w tym rozdziale powrócimy do tych zagadnień. Będzie to nieco inne spojrzenie na obiekty iterowalne w kontekście wprowadzenia do tematyki *funkcji generatorów* i powiązanych z nimi *wyrażeń generatorów*, pozwalających na budowanie przez programistę obiektów generujących wyniki na żądanie.

Iteracje w Pythonie mogą być również obsługiwane przez *klasy* definiowane przez użytkownika, ale omawianie tych zagadnień odłożymy do części VI, w której zajmiemy się przeciążaniem operatorów. Ponieważ niniejszym rozdziałem zamykamy omawianie wbudowanych mechanizmów iteracyjnych, podsumujemy tutaj różne narzędzia, które poznaliśmy do tej pory. W następnym rozdziale będziemy kontynuować ten wątek, mierząc względną wydajność tych narzędzi w nieco bardziej rozbudowanym studium przypadku. Zanim to jednak nastąpi, w tym rozdziale powrócimy do historii związanej z listami składanymi oraz iteracjami i rozszerzmy ją o generatory wartości.

Listy składane i narzędzia funkcyjne

Jak wspominaliśmy na początku tej książki, Python obsługuje paradygmaty programowania proceduralnego, obiektowego i funkcyjnego. W rzeczywistości Python posiada wiele narzędzi uznawanych przez większość użytkowników za *funkcyjne* z natury, które wymieniliśmy w poprzednim rozdziale — domknięcia, generatory, wyrażenia lambda, złożenia, odwzorowania, dekoratory, obiekty funkcji i wiele innych. Narzędzia te pozwalają nam nie tylko stosować i łączyć funkcje w rozwiązywaniu dające ogromne możliwości, ale często również oferują możliwość zapamiętywania stanów i sposoby kodowania, które są alternatywą dla klas i programowania zorientowanego obiektowo.

Na przykład w poprzednim rozdziale omawialiśmy narzędzia `map` i `filter` — kluczowe elementy wczesnego zestawu narzędzi programistycznych Pythona inspirowanych językiem Lisp — które mapują operacje na obiektach iterowalnych i gromadzą wyniki ich działania. Tego typu schemat operacji jest na tyle powszechny w Pythonie, że z czasem zaimplementowano w nim nowy typ wyrażeń: *listy składane*, które są jeszcze bardziej elastyczne niż narzędzia do tej pory przez nas poznawane.

Według historii języka Python listy składane były pierwotnie inspirowane podobnym narzędziem w funkcyjnym języku programowania Haskell mniej więcej w czasach Pythona 2.0. W skrócie: zamiast stosować funkcje, listy składane używają dowolnych wyrażeń do elementów obiektów iterowalnych, dzięki czemu mogą być znacznie bardziej uniwersalnymi narzędziami. W późniejszych wersjach Pythona złożenia zostały rozszerzone na inne komponenty: zbiory, słowniki, a nawet wyrażenia generatorów wartości, które omówimy w tym rozdziale — jak widać złożenia nie dotyczą już tylko list.

Z zagadnieniem list składanych spotkaliśmy się już na chwilę w rozdziale 4. i omówiliśmy je nieco szerzej w rozdziale 14. w powiązaniu z instrukcjami pętli. Ponieważ są one również powiązane z funkcyjnymi narzędziami programistycznymi, takimi jak wywołania `map` i `filter`, wróćmy tutaj do tego tematu po raz ostatni. Technicznie rzecz biorąc, mechanizm ten nie jest powiązany z funkcjami — jak zobaczymy, listy składane mogą być bardziej ogólnym narzędziem niż funkcje `map` i `filter` — ale czasami najłatwiej jest je zrozumieć poprzez analogię do alternatyw opartych na funkcjach.

Listy składane kontra funkcja `map`

Przyjrzyjmy się przykładowi, który demonstruje podstawowe zastosowania. Jak widzieliśmy w rozdziale 7., funkcja wbudowana `ord` zwraca kod liczbowy przekazanego znaku ASCII (jej odwrotnością jest funkcja `chr`, zwracająca znak na podstawie jego kodu ASCII). Zwracane liczby odpowiadają kodom ASCII znaków, o ile poszczególne znaki mieszczą się w 7-bitowym zakresie kodów zestawu znaków ASCII:

```
>>> ord('s')
```

```
115
```

Załóżmy, że chcemy odczytać kody ASCII *wszystkich* znaków podanego ciągu. Najprostsze podejście polega na użyciu prostej pętli `for`, która po kolei zapisze odpowiednie kody na liście:

```
>>> res = []
```

```
>>> for x in 'spam':
```

```
    res.append(ord(x))          # Ręczne zbieranie wyników
```

```
>>> res
```

```
[115, 112, 97, 109]
```

Jednak gdy znamy już funkcję `map`, możemy pokusić się o osiągnięcie tego samego rezultatu w jednym wywołaniu, bez konieczności budowania listy w kodzie.

```
>>> res = list(map(ord, 'spam'))      # Wywołanie funkcji na sekwencji  
(lub innym obiekcie iterowalnym)
```

```
>>> res
```

```
[115, 112, 97, 109]
```

Jednak identyczny wynik osiągniemy, stosując wyrażenie listy składanej. Funkcja `map` dokonuje odwzorowania *funkcji* na obiekcie iterowalnym, natomiast lista składana dokonuje odwzorowania *wyrażenia* na sekwencji lub innym obiekcie iterowalnym.

```
>>> res = [ord(x) for x in 'spam']      # Wywołanie wyrażenia na sekwencji  
(lub innym obiekcie iterowalnym)
```

```
>>> res
```

```
[115, 112, 97, 109]
```

Listy składane gromadzą wyniki wywołania dowolnego wyrażenia na sekwencji wartości i zwracają nową sekwencję. Składnia list składanych wymaga użycia nawiasów kwadratowych (co sugeruje, że tworzą listę). W prostej wersji wewnętrznych nawiasów kwadratowych definiuje się wyrażenie składające się z nazwy zmiennej, po której następuje coś, co przypomina uproszczenie deklaracji pętli `for`, w którym jest użyta ta sama zmienna. Python zgromadzi wyniki wykonania wyrażenia na elementach odczytanych z sekwencji w wewnętrznej pętli `for`.

Wynik wykonania powyższego wyrażenia przypomina wynik zastosowania pętli `for` lub funkcji `map`. Listy składane stają się wygodniejsze wówczas, gdy do obiektu iterowalnego zamiast funkcji chcemy zastosować bardziej złożone wyrażenia.

```
>>> [x ** 2 for x in range(10)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

W powyższym przykładzie obliczamy kwadraty liczb od 0 do 9 (pozwoliliśmy tutaj na wyświetlenie wyniku w sesji interaktywnej, ale możemy go przypisać do zmiennej, jeżeli chcemy go zachować i użyć w dalszej części programu). Aby uzyskać podobny efekt za pomocą funkcji `map`, musielibyśmy sami napisać prostą funkcję obliczającą kwadrat podanej liczby. Jeżeli funkcja byłaby potrzebna tylko w tym jednym miejscu, moglibyśmy zdefiniować ją w miejscu za pomocą wyrażenia `lambda`, zamiast kodować nazwaną funkcję za pomocą instrukcji `def`.

```
>>> list(map((lambda x: x ** 2), range(10)))  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Powyższy kod wykonuje dokładnie to samo zadanie i jest zaledwie o kilka znaków dłuższy od odpowiadającej mu wersji używającej listy składanej. Jest też niewiele bardziej skomplikowany (oczywiście pod warunkiem że ktoś rozumie działanie instrukcji `lambda`). Jednak w przypadku bardziej skomplikowanych wyrażeń, listy składane często pozwalają zaoszczędzić sporo pisania. Zademonstrujemy to już w kolejnym podrozdziale.

Dodajemy warunki i pętle zagnieżdżone — `filter`

Listy składane są narzędziem jeszcze bardziej ogólnym, niż pokazaliśmy to dotychczas. Na przykład, jak to pokazywaliśmy w rozdziale 14., w wyrażeniu listy składanej po deklaracji pętli `for` możemy dodać warunek `if`, który uzupełnia nasze wyrażenie o logikę wyboru danych. Listy składane zawierające warunek `if` można traktować jako odpowiednik funkcji `filter` omówionej w poprzednim rozdziale: pozwalają pominąć te elementy sekwencji, dla których warunek nie jest spełniony.

Zademonstrujmy obydwa podejścia na przykładzie wyboru liczb parzystych z sekwencji liczb od 0 do 4. Podobnie jak w powyższym przykładzie z funkcją `map`, wersja wykorzystująca funkcję `filter` do obsługi warunku używa prostej funkcji `lambda`. Dla kompletności przedstawiamy też odpowiednią wersję z pętlą `for`.

```
>>> [x for x in range(5) if x % 2 == 0]  
[0, 2, 4]  
>>> list(filter((lambda x: x % 2 == 0), range(5)))  
[0, 2, 4]  
>>> res = []  
>>> for x in range(5):  
    if x % 2 == 0:  
        res.append(x)  
>>> res  
[0, 2, 4]
```

Wszystkie wersje do wykrywania liczb parzystych wykorzystują operację `modulo` (reszta z dzielenia całkowitego, operator `%`): jeżeli dzielenie przez 2 nie ma reszty, to dzielna jest liczbą

parzystą. Wersja wykorzystująca funkcję `filter` jest niewiele dłuższa od wersji wykorzystującej listę składaną. Jednak w przypadku list składanych możemy wykorzystać skomplikowane wyrażenie oraz warunek wyboru danych z sekwencji, co w efekcie zadziała jak złożenie w jednym wyrażeniu funkcji `map` i `filter`.

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

Tym razem wynik ma zawierać kwadraty liczb parzystych z zakresu od 0 do 9 — wewnętrzna pętla `for` pomija liczby niespełniające warunku parzystości, a wyrażenie po lewej stronie `for` oblicza wynik. Analogiczne wywołanie funkcji `map` wymagałoby znacznie więcej pracy: musielibyśmy wykonać złożenie wywołań funkcji `map` z funkcją `filter`, co w efekcie da o wiele bardziej skomplikowane wyrażenie:

```
>>> list( map((lambda x: x**2), filter((lambda x: x % 2 == 0), range(10))) )
[0, 4, 16, 36, 64]
```

Formalna składnia list składanych

W rzeczywistości listy składane są jeszcze bardziej ogólnym narzędziem. W najprostszej postaci listy składanej zawsze musimy zakodować wyrażenie akumulujące i pojedynczą klauzulę `for`:

```
[wyrażenie for cel in obiekt_iterowalny]
```

Chociaż wszystkie inne części składni takiej listy są opcjonalne, pozwalają na definiowanie bardziej złożonych iteracji w listach składanych, możesz więc zakodować dowolną liczbę zagnieżdżonych pętli, a każda z nich może mieć opcjonalną powiązaną klauzulę `if`, działającą jak filtr. Ogólna struktura listy składanej wygląda następująco:

```
[wyrażenie for zmienna1 in obiekt_iterowalny1 [if warunek1]
          for zmienna2 in obiekt_iterowalny2 [if warunek2] ...
          for zmiennaN in obiekt_iterowalnyN [if warunekN]]
```

Ta sama składnia jest dziedziczona przez zbiory i słowniki oraz *wyrażenia generatorów* (o których będziemy mówić już niebawem), chociaż używają one różnych znaków zamykających (nawiasy klamrowe lub często opcjonalne nawiasy zwykłe), a słowniki składane rozpoczynają się od dwóch wyrażeń oddzielonych dwukropkiem (dla klucza i wartości).

W poprzednim rozdziale eksperymentowaliśmy z klauzulą filtrującą `if`. Kiedy w wyrażeniu listy składanej *zagnieździmy* kolejne klauzule `for`, będą one działały analogicznie do zagnieżdżonych pętli `for`, na przykład:

```
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Wyrażenie to daje taki sam wynik jak ten następujący, znacznie bardziej rozbudowany odpowiednik:

```
>>> res = []
>>> for x in [0, 1, 2]:
          for y in [100, 200, 300]:
              res.append(x + y)
>>> res
```

```
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Choć listy składane zwracają listy, należy pamiętać, że mogą działać na dowolnej sekwencji lub innym obiekcie iterowalnym. Poniższy przykład wykonuje złożenie listy na podstawie znaków z ciągów zamiast sekwencji liczb i zwraca kombinacje par znaków z każdego ciągu:

```
>>> [x + y for x in 'spam' for y in 'SPAM']  
['sS', 'sP', 'sA', 'sM', 'pS', 'pP', 'pA', 'pM',  
'aS', 'aP', 'aA', 'aM', 'mS', 'mP', 'mA', 'mM']
```

Każda klauzula `for` może mieć przypisany filtr `if`, bez względu na to, jak głęboko zagnieżdżone są pętle — chociaż przy większych stopniach zagnieżdżenia coraz trudniej sobie wyobrazić potencjalne zastosowania dla takiego kodu, być może oprócz zaawansowanego przetwarzania tablic wielowymiarowych:

```
>>> [x + y for x in 'spam' if x in 'sm' for y in 'SPAM' if y in ('P', 'A')]  
['sP', 'sA', 'mP', 'mA']  
>>> [x + y + z for x in 'spam' if x in 'sm'  
      for y in 'SPAM' if y in ('P', 'A')  
      for z in '123' if z > '1']  
['sP2', 'sP3', 'sA2', 'sA3', 'mP2', 'mP3', 'mA2', 'mA3']
```

Na koniec przedstawiamy jeszcze jeden, bardziej skomplikowany przykład listy składanej, ilustrujący efekt użycia warunków `if` w zagnieżdżonych klauzulach `for`, zastosowanych do obiektów numerycznych, a nie jak do tej pory do ciągów znaków:

```
>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]  
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Powyższe wyrażenie zwraca permutacje liczb parzystych z zakresu od 0 do 4 z liczbami nieparzystymi z tego zakresu. Warunki `if` odfiltrowują zbędne elementy w każdym z podwyrażeń. Wyrażenie to odpowiada następującemu kodowi z użyciem zagnieżdżeń pętli `for`:

```
>>> res = []  
>>> for x in range(5):  
    if x % 2 == 0:  
        for y in range(5):  
            if y % 2 == 1:  
                res.append((x, y))  
>>> res  
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Warto przypomnieć, że jeżeli nie do końca rozumiesz, co robi złożone wyrażenie listy składanej, zawsze możesz rozpisać zagnieżdżone klauzule w kilku wierszach kodu z użyciem odpowiednio dobranych wcięć, aby uzyskać równoważną instrukcję. Rezultat będzie dłuższy, ale na pierwszy rzut oka bardziej czytelny, szczególnie dla tych mniej doświadczonych użytkowników.

Wersja tego ostatniego przykładu, ale przygotowana z użyciem funkcji `map` i `filter` będzie znacznie dłuższa i bardziej skomplikowana, z dużą liczbą zagnieżdżeń, zatem nie będziemy jej

tutaj demonstrować i pozostawimy ją jako ćwiczenie do samodzielnego wykonania dla mistrzów zen, byłych programistów Lispa i nawiedzonych maniaków programowania.

Przykład — listy składane i macierze

Oczywiście nie wszystkie zastosowania list składanych są tak akademickie. Przyjrzyjmy się zatem kilku zastosowaniom, które wymagają wytyżenia szarych komórek. Jak mogłeś się przekonać w rozdziałach 4. i 8., jednym ze sposobów implementacji macierzy (czy inaczej mówiąc, tablic wielowymiarowych) w Pythonie jest zagnieżdżona struktura list. Poniższy przykład implementuje macierz 3×3 w postaci listy zagnieżdżonych list:

```
>>> M = [[1, 2, 3],  
           [4, 5, 6],  
           [7, 8, 9]]  
  
>>> N = [[2, 2, 2],  
           [3, 3, 3],  
           [4, 4, 4]]
```

W tego typu strukturze możemy nawigować, podając indeks wiersza i indeks kolumny w wierszu:

```
>>> M[1]                                # Wiersz 2  
[4, 5, 6]  
  
>>> M[1][2]                            # Wiersz 2, element 3  
6
```

Listy składane są doskonałym narzędziem do przetwarzania struktur tego typu, ponieważ pozwalają przeszukiwać wiersze i kolumny w sposób automatyczny. Mimo, że nasza struktura przechowuje macierz w postaci listy wierszy, aby wydobyć tylko drugą *kolumnę*, listy składane mogą przeiterować po wierszach, wydobywając kolumnę po indeksie, albo wręcz wykorzystać indeksy wierszy i indeks odpowiedniej kolumny:

```
>>> [row[1] for row in M]                # Kolumna 2  
[2, 5, 8]  
  
>>> [M[row][1] for row in (0, 1, 2)]    # Zastosowanie offsetów  
[2, 5, 8]
```

Dzięki indeksom możemy wykonywać bardziej skomplikowane operacje, jak na przykład wydobywanie *przekątnej*. Pierwsze z poniższych wyrażeń do wygenerowania sekwencji współrzędnych wykorzystuje funkcję `range`, a następnie z macierzy wydobywa wiersz i kolumnę o tej samej współrzędnej, na przykład `M[0][0]`, następnie `M[1][1]` i tak dalej. Drugie wyrażenie skaluje indeks kolumn, aby pobrać elementy `M[0][2]`, `M[1][1]` itd. (zakładamy, że macierz jest kwadratowa).

```
>>> [M[i][i] for i in range(len(M))]      # Przekątne  
[1, 5, 9]  
  
>>> [M[i][len(M)-1-i] for i in range(len(M))]  
[3, 5, 7]
```

Zmiana takiej macierzy *w miejscu* wymaga przypisania do offsetów (jeżeli macierz nie jest kwadratowa, musimy dwukrotnie użyć funkcji `range`):

```
>>> L = [[1, 2, 3], [4, 5, 6]]  
>>> for i in range(len(L)):  
    for j in range(len(L[i])):          # Aktualizacja w miejscu  
        L[i][j] += 10  
>>> L  
[[11, 12, 13], [14, 15, 16]]
```

Nie możemy zrobić tego samego za pomocą list składanych, ponieważ tworzą one *nowe* listy, ale zawsze możemy przypisać ich wyniki do oryginalnej nazwy, aby uzyskać podobny efekt. Na przykład możemy zastosować wybraną operację do każdego elementu w macierzy i uzyskać wyniki w prostym wektorze lub macierzy o tym samym kształcie:

```
>>> [col + 10 for row in M for col in row]      # Przypisanie do M, aby  
zachować nową wartość  
[11, 12, 13, 14, 15, 16, 17, 18, 19]  
>>> [[col + 10 for col in row] for row in M]  
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

Aby łatwiej Ci było zrozumieć powyższe polecenia, spróbujemy przetłumaczyć je na odpowiadające im proste polecenia przedstawione poniżej — zastosujemy odpowiednie wcięcia dla kolejnych elementów znajdujących się dalej w prawo w oryginalnym wyrażeniu (jak w pierwszej pętli poniżej) i utworzymy nową listę, gdy wyrażenia są zagnieżdżone po lewej stronie (np. druga pętla poniżej). Jak łatwo zauważyc w poniższym kodzie, drugie wyrażenie z przykładu powyżej działa, ponieważ iteracja wiersza jest pętlą zewnętrzna: dla każdego wiersza uruchamia iterację zagnieżdzonej kolumny, aby utworzyć jeden wiersz macierzy wynikowej:

```
>>> res = []  
>>> for row in M:                         # Odpowiedniki polecień  
    for col in row:                        # Wcinamy kolejne części w  
prawo  
        res.append(col + 10)  
  
>>> res  
[11, 12, 13, 14, 15, 16, 17, 18, 19]  
>>> res = []  
>>> for row in M:  
    tmp = []                                # Zagnieżdżanie po lewej  
rozpoczyna nową listę  
    for col in row:  
        tmp.append(col + 10)  
    res.append(tmp)  
  
>>> res  
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

Wreszcie przy odrobinie kreatywności listy składane mogą posłużyć też do wykonywania operacji na *wielu macierzach*. W poniższym przykładzie najpierw budujemy płaską listę, zawierającą wyniki mnożenia odpowiednich par wartości z obu macierzy, a następnie tworzymy zagnieżdżoną strukturę list z tymi wartościami poprzez ponowne zagnieżdżanie list składanych:

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> N
[[2, 2, 2], [3, 3, 3], [4, 4, 4]]
>>> [M[row][col] * N[row][col] for row in range(3) for col in range(3)]
[2, 4, 6, 12, 15, 18, 28, 32, 36]
>>> [[M[row][col] * N[row][col] for col in range(3)] for row in range(3)]
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Ostatnie wyrażenie działa, ponieważ iteracja wierszy odbywa się w pętli zewnętrznej; odpowiednikiem tego rozwiązania z użyciem zagnieżdżonych pętli `for` jest kod przedstawiony poniżej:

```
>>> res = []
>>> for row in range(3):
    tmp = []
    for col in range(3):
        tmp.append(M[row][col] * N[row][col])
    res.append(tmp)
```

Dla zabawy możemy użyć funkcji `zip` do sparowania elementów, które będą zwielenokrotniane — przedstawione poniżej złożenia i pętle dają ten sam wynik mnożenia elementów list jak w poprzednim przykładzie (a ponieważ funkcja `zip` jest generatorem wartości w wersji 3.x, takie rozwiązanie nie jest wcale tak nieefektywne, jak mogłoby się na pierwszy rzut oka wydawać):

```
[[col1 * col2 for (col1, col2) in zip(row1, row2)] for (row1, row2) in zip(M, N)]
res = []
for (row1, row2) in zip(M, N):
    tmp = []
    for (col1, col2) in zip(row1, row2):
        tmp.append(col1 * col2)
    res.append(tmp)
```

W porównaniu ze swoim odpowiednikiem napisanym z użyciem funkcji wersja z użyciem list składanych wymagała napisania tylko jednego wiersza kodu, a w przypadku dużych macierzy będzie działała znacznie szybciej. Dalszą część tej historii opowiemy już w kolejnym podrozdziale.

Nie nadużywaj list składanych: reguła KISS

Listy składane działają na takim stopniu uogólnienia, że szybko mogą stać się niezrozumiałe, szczególnie kiedy używamy wielu poziomów zagnieżdżeń. Niektóre zadania programistyczne są z natury skomplikowane i nie możemy zrobić nic, aby uczynić je prostszymi (doskonałym przykładem mogą być permutacje, o których będziemy mówić już za chwilę). Rozwiązań takie jak listy składane są potężnymi narzędziami, jeżeli są mądrze stosowane, i nie ma niczego złego w ich używaniu w skryptach.

Warto zauważyć, że kod podobny do tego z poprzedniej sekcji może zwiększyć złożoność skryptu bardziej, niż powinien, i szczerze mówiąc, ma tendencję do nadmiernego wzbudzania zainteresowania mniej doświadczonych programistów, wychodzących z błędnego założenia, że tworzenie skomplikowanego kodu w jakiś sposób podkreśla ich talent. Ponieważ dla niektórych użytkowników takie narzędzia są bardziej atrakcyjne, niż powinny, spróbujemy jasno określić ich zakres zastosowań.

W naszej książce prezentujemy różne formy mniej lub bardziej zaawansowanych złożień, ale w prawdziwym świecie tworzenie skomplikowanego i trudnego do zrozumienia kodu, kiedy nie jest to uzasadnione, świadczy po prostu o niezbyt dobrej inżynierii oprogramowania i wątpliwych umiejętnościach programisty. Chciałbym jeszcze raz podkreślić to, o czym pisaliśmy w pierwszym rozdziale: celem programowania nie jest tworzenie bardzo mądrego i niezrozumiałego kodu — chodzi o to, jak jasno Twój program komunikuje cel swojego działania.

Zamiast tego możemy również przytoczyć wielokrotnie już cytowane motto Pythona:

Proste jest zawsze lepsze niż złożone.

Celowe tworzenie nadmiernie skomplikowanego kodu może być fajnym, akademickim ćwiczeniem, ale nie powinno mieć miejsca w programach, których kod będzie kiedyś analizowany i używany przez innych użytkowników.

Moja rada jest taka, aby poczynający użytkownicy Pythona pisali tego typu kod w postaci pętli `for`, a funkcje `map` lub listy składane wykorzystywali wówczas, gdy są one łatwe w użyciu. Jak zawsze ma tutaj zastosowanie zasada mówiąca, że „piękno tkwi w prostocie” — zwięzłość kodu jest o wiele mniej ważna od jego czytelności i przejrzystości. Innymi słowy, stara, dobra reguła KISS nadal obowiązuje (ang. *Keep It Simple, Stupid*), zatem jeżeli możesz, zrób to w prosty, skuteczny i zawsze działający sposób.

Druga strona medalu: wydajność, zwięzłość, ekspresyjność

Jednak w przypadku list składanych dodatkowy poziom komplikacji pozwala uzyskać spore korzyści w postaci wzrostu wydajności; przeprowadzone testy wykazują dwukrotnie większą wydajność wywołań `map` w porównaniu z pętlami realizującymi te same zadania, a listy składane są często jeszcze wydajniejsze od funkcji `map`. Tego typu rozbieżności w wydajności działania różnią się w zależności od wersji Pythona i konkretnej implementacji danego zadania, ale ogólnie biorą się z tego, że funkcja `map` i listy składane są wykonywane w samym interpreterze z wydajnością języka C, co jest znacznie szybsze niż kod bajtowy pętli `for` wykonywany przez maszynę wirtualną Pythona.

Oprócz tego listy składane oferują większą zwięzłość kodu, która jest przekonującą, a nawet uzasadniona, gdy to zmniejszenie rozmiaru nie oznacza również utraty przejrzystości i utrudnienia czytelności kodu dla następnego programisty. Co więcej, wielu uważa ekspresyjność list składanych za potężnego sojusznika. Ponieważ funkcja `map` i listy składane są wyrażeniami, mogą one się pojawiać syntaktycznie w miejscach, w których nie możemy używać instrukcji pętli `for`, na przykład w ciałach funkcji `lambda`, w literałach list i słowników itp.

Z tego powodu listy składane i wywołania funkcji `map` są warte poznania i używania w przypadku prostszych rodzajów iteracji, zwłaszcza jeżeli szybkość działania aplikacji jest czynnikiem krytycznym. Nie zmienia to jednak w niczym faktu, że użycie w zamian pętli `for` znacząco zwiększa czytelność kodu i ułatwia analizę jego sposobu działania, toteż są one zalecanym rozwiązaniem ze względu na swoją prostotę. Korzystając z takich rozwiązań,

powinieneś starać się, aby wywołania funkcji `map` i listy składane były proste i przejrzyste; w przypadku bardziej złożonych zadań powinieneś raczej używać implementacji z użyciem pętli.



Jak już wspominaliśmy, uogólnione opinie na temat *wydajności* poszczególnych rozwiązań, takie jak przedstawione powyżej, mogą zależeć od sposobów wywoływanego funkcji, a także zmian i optymalizacji w poszczególnych wersjach Pythona. Na przykład w najnowszych wersjach Pythona znacząco przyspieszone zostało działanie pętli `for`, co nie zmienia faktu, że w wielu zastosowaniach listy składane są nadal znacznie szybsze niż pętle czy nawet wywołania funkcji `map`, choć to ostatnie rozwiązanie zwykle nadal nie ma sobie równych, gdy do poszczególnych elementów obiektu iterowalnego musimy zastosować wywołanie określonej funkcji. Krótko mówiąc, w kolejnym rozdziale postaramy się udowodnić nasze tezy dotyczące szybkości działania pętli, wywołań funkcji `map` i list składanych i nasze rozważania pozostaną aktualne przynajmniej do czasu, kiedy w pythonowej implementacji tych mechanizmów nie nastąpią jakieś radykalne zmiany. Jeżeli chcesz samodzielnie sprawdzić wydajność poszczególnych rozwiązań, powinieneś zatrzymać się w zestawie narzędzi dostępnego w standardowej bibliotece `time` lub w nowszym module `timeit` (dodanym w wersji 2.4) albo zapoznać się z rozszerzonym opisem tych rozwiązań, który znajdziesz w następnym rozdziale.

Funkcje i wyrażenia generatorów

Python jest obecnie znacznie bardziej oszczędny niż dawniej: oferuje mnóstwo narzędzi, które generują wyniki tylko wówczas, gdy te są potrzebne, a nie wszystkie naraz. Widzieliśmy to w działaniu, gdy korzystaliśmy z wielu wbudowanych narzędzi: plików, które na żądanie odczytują wiersze, czy funkcji takich jak `map` i `zip`, które w wersji 3.x tworzą określone elementy na żądanie. Takie „oszczędności” nie ograniczają się jednak do samego Pythona. Do tej kategorii narzędzi należy w szczególności zaliczyć dwie konstrukcje, które opóźniają generowanie wyników, gdy tylko jest to możliwe, w operacjach zdefiniowanych przez użytkownika:

- *funkcje generatorów* (dostępne od wersji 2.3), które tworzymy tak samo jak zwykłe funkcje z użyciem instrukcji `def`, z tym że do zwracania wyników cząstkowych wykorzystujemy instrukcję `yield`, która zwraca wyniki działania po jednym na raz i zawiesza wykonanie funkcji, zachowując jej stan, co pozwala na jej wznowienie w przypadku, gdy odbiorca znów poprosi o dane;
- *wyrażenia generatorów* (dostępne od wersji 2.4) mają składnię zbliżoną do list składanych opisywanych w poprzedniej sekcji, ale zwracają obiekt generujący wyniki cząstkowe na żądanie, zamiast tworzyć naraz całą listę wyników.

Warto pamiętać: listy składane oraz funkcja `map`

Oto bardziej realistyczne przykłady zastosowania list składanych i funkcji `map` (opisywany problem został już rozwiązany z użyciem list składanych w rozdziale 14., ale wracamy do niego, aby uzupełnić go o zastosowanie funkcji `map`). Jak pamiętasz, metoda `readlines` obiektu plikowego zwraca wiersze pliku zakończone znakiem końca wiersza `\n` (w kodzie przedstawionym w przykładzie poniżej założono, że w bieżącym katalogu roboczym znajduje się plik zawierający trzy wiersze tekstu):

```
>>> open('myfile').readlines()
['aaa\n', 'bbb\n', 'ccc\n']
```

Jeżeli chcesz się pozbyć znaku końca wiersza, możesz tego dokonać z użyciem list składanych lub funkcji `map` (wynikiem działania funkcji `map` w wersji 3.x jest obiekt

iterowalny, zatem w celu wyświetlenia na ekranie musisz opakować go w wywołanie funkcji `list`):

```
>>> [line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']
>>> [line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']
>>> list(map((lambda line: line.rstrip()), open('myfile')))
['aaa', 'bbb', 'ccc']
```

Ostatnie dwie wersje wykorzystują *iteratory plikowe* (o których pisaliśmy w rozdziale 14.), co w praktyce oznacza, że do odczytu zawartości pliku nie musimy jawnie wywoływać jego metod. Wywołanie funkcji `map` wymaga wpisania nieco większej liczby znaków w porównaniu z listą składaną, ale dzięki temu nie musimy bezpośrednio tworzyć wyniku w postaci listy.

Listy składane mogą być również użyte w charakterze operacji wydobywającej kolumny z bazy danych. Standardowe API Pythona do komunikacji z *bazami danych SQL* zwraca wyniki zapytań w postaci list krotek — lista jest odwzorowaniem tabeli, krotki są wierszami, a elementy krotek są wartościami w kolumnach:

```
>>> listoftuple = [('Teodor', 35, 'dyrektor'), ('Teofil', 40, 'prezes')]
```

Dane z poszczególnych kolumn można odczytać za pomocą pętli `for`, ale funkcja `map` i listy składane pozwalają wykonać to w jednym wierszu kodu, a do tego szybciej.

```
>>> [age for (name, age, job) in listoftuple]
[35, 40]
>>> list(map((lambda row: row[1]), listoftuple))
[35, 40]
```

Pierwsze wyrażenie wykorzystuje przypisanie do *krotki zmiennych*, co pozwala rozpakować trzyelementową krotkę na trzy zmienne, drugie podejście wykorzystuje odczyt wartości po indeksie. W Pythonie 2.x (ale nie w 3.x — patrz uwaga dotycząca rozpakowywania listy argumentów funkcji w rozdziale 18.) funkcja `map` również może rozpakować listę swoich argumentów.

```
# tylko dla 2.x
>>> list(map((lambda (name, age, job): age), listoftuple))
[35, 40]
```

Informacje na temat API dostępu do baz danych dla Pythona można znaleźć w licznych książkach i innych źródłach dokumentacji Pythona.

Oprócz rozróżnienia między uruchamianiem funkcji a wyrażeniami, największa różnica między funkcją `map` a listami składanymi w Pythonie 3.x polega na tym, że funkcja `map` jest *obiektem iterowalnym*, generującym wyniki na żądanie. Aby uzyskać tę samą skalę oszczędności pamięci i szybkości działania, należałoby listę składaną zapisać jako *wyrażenie generatora* (to jeden z tematów, do których przejdziemy już za chwilę).

Ponieważ żadna z wymienionych wyżej konstrukcji języka nie zwraca pełnej listy wyników naraz, pozwala to na zmniejszenie zajętości pamięci oraz podzielenie czasu wykorzystania procesora pomiędzy kolejne wywołania. Jak się wkrótce przekonasz, oba rozwiązania swoje działanie implementują w oparciu o *protokół iteracji*, który poznaliśmy w rozdziale 14.

Takie mechanizmy nie są nowe (funkcje generatora były dostępne jako opcja już w Pythonie 2.2) i są dość powszechnie wykorzystywane w dzisiejszym kodzie Pythona. Pojęcie generatorów w Pythonie wiele zawdzięcza innym językom programowania, zwłaszcza językowi Icon. Jeżeli jesteś przyzwyczajony do prostszych modeli programowania, początkowo generatory mogą wydawać Ci się niezwykłe, jednak bardzo szybko przekonasz się, że w odpowiednich zastosowaniach jest to naprawdę potężne narzędzie. Co więcej, ponieważ są naturalnym rozszerzeniem funkcji, złożień i iteracji, które już omawialiśmy, o tworzeniu generatorów wiesz już więcej, niż możesz się spodziewać.

Funkcje generatorów – yield kontra return

W tej części książki dowiedzieliśmy się, w jaki sposób kodować zwykłe funkcje otrzymujące parametry wejściowe i zwracające cały wynik w jednej instrukcji `return`. W Pythonie istnieje jednak możliwość napisania funkcji, która może zwracać wartość cząstkową, zapisując swój stan, aby po wznowieniu kontynuować od tego miejsca, w którym zatrzymała się poprzednio. Tego typu funkcje, dostępne zarówno w Pythonie 2.x, jak i 3.x, znane są jako *funkcje generatorów*, ponieważ generują sekwencję wyników stopniowo, nie w jednej całości.

Funkcje generatora pod wieloma względami przypominają normalne funkcje i faktycznie są kodowane za pomocą normalnych instrukcji `def`. Jednak po utworzeniu są one specjalnie komplikowane do postaci obiektu, który obsługuje protokół iteracji, a po wywołaniu nie zwracają wyniku, tylko generator wyników, który może pojawić się w dowolnym kontekście iteracji. Obiekty iterowalne omawialiśmy już w rozdziale 14., a na rysunku 14.1 pokazaliśmy formalne i graficzne podsumowanie sposobu ich działania. W tym rozdziale ponownie zajmiemy się tymi zagadnieniami, aby zbadać ich związek z generatorami.

Zawieszanie stanu

W przeciwieństwie do zwykłych funkcji zwracających wartość i kończących tym samym swoje działanie funkcje generatorów potrafią zawieszać swoje działanie i wznowiać je sekwencyjnie w miarę generowania kolejnych wartości. Dzięki temu są często użyteczną alternatywą dla tworzenia wyników w całości, na przykład dla klas, w których programista samodzielnie obsługuje zapisywanie i odtwarzanie stanu. Stan, który zachowują funkcje generatorów podczas zawieszenia, obejmuje zarówno miejsce w kodzie, jak i cały zasięg lokalny, stąd ich *zmienne lokalne* przechowują informacje między kolejnymi wywołaniami i udostępniają je po wznowieniu działania funkcji.

Podstawowa różnica między generatorami a zwykłymi funkcjami polega na tym, że generator zwraca wynik za pomocą instrukcji `yield`, a nie `return`. Instrukcja `yield` zawiesza wykonanie funkcji i zwraca wynik do odbiorcy (kodu wywołującego), ale zachowuje stan wykonania, dzięki czemu funkcja może być wznowiona od tego samego miejsca. Po wznowieniu funkcja kontynuuje działanie natychmiast po wykonanej poprzednio instrukcji `yield`. Z punktu widzenia funkcji mamy możliwość tworzenia szeregu wartości po jednej zamiast obliczania wszystkich za jednym zamachem i zwracania w całości, na przykład w postaci listy.

Integracja protokołu iteracji

Aby w pełni zrozumieć funkcje generatorów, należy poznać ich związek z protokołem iteracyjnym Pythona. Jak już wiemy, obiekty iteratorów posiadają metodę `__next__` (w wersji 2.x ta metoda nosi nazwę `next`), która zwraca kolejny element iteracji lub wywołuje wyjątek `StopIteration`. Iterator obiektu iterowalnego jest zwracany po wywołaniu funkcji wbudowanej `iter`, choć krok ten nie jest obowiązkowy dla obiektów, które są swoimi własnymi iteratorami.

Pętle `for` w Pythonie oraz inne konteksty iteracyjne wykorzystują protokół iteracji do odczytu kolejnych elementów sekwencji lub wartości zwracanych przez generator, jeżeli ten protokół jest obsługiwany (jeżeli nie, iteracja przechodzi w odczyt statycznej sekwencji w kolejności elementów). Każdy obiekt obsługujący ten interfejs działa we wszystkich narzędziach iteracyjnych.

W celu obsługi protokołu iteratorów instrukcje `yield` są komplikowane jako *generatory* — nie są to normalne funkcje, ponieważ są budowane w celu zwracenia obiektu posiadającego odpowiednie metody protokołu iteracji. Po wywołaniu zwracają obiekt generatora obsługujący interfejs iteracji, który z kolei obsługuje automatycznie utworzoną metodę `__next__` wznowiającą wykonanie funkcji.

Funkcje generatorów mogą również obsługiwać instrukcję `return`, która znajduje się zwykle na końcu bloku `def` i powoduje zakończenie działania funkcji, jak również sygnalizuje zakończenie generowania kolejnych wartości — technicznie rzecz biorąc, w takiej sytuacji po każdym normalnym zwróceniu wyniku wywoływany jest wyjątek `StopIteration`. Z punktu widzenia kodu wywołującego metoda `__next__` generatora wznowia wykonanie funkcji aż do następnej instrukcji `yield` lub wywołania wyjątku `StopIteration`.

Najważniejszym spostrzeżeniem jest to, że funkcje generatorów kodowane w instrukcji `def` zawierają instrukcję `yield` i automatycznie obsługują protokół iteracji, dzięki czemu mogą być używane w dowolnym kontekście iteracyjnym do generowania wyników na żądanie.



Jak wspomniałem w rozdziale 14., w Pythonie 2.x i wcześniejszych obiekty iterowalne definiują metody `next`. Dotyczy to również obiektów generatorów omówionych w tym rozdziale. W Pythonie 3.x nazwa metody `next` została zmieniona na `__next__`. Wbudowana funkcja `next` jest udostępniana jako narzędzie zwiększające wygodę programowania i ułatwiające tworzenie przenośnego kodu: wywołanie `next(I)` działa tak samo jak `I.__next__()` w 3.x i `I.next()` w 2.6 i 2.7. W wersjach wcześniejszych od 2.6 należy po prostu używać metody `I.next()`, zamiast ręcznie iterować po elementach.

Funkcje generatorów w działaniu

W celu zilustrowania podstaw działania generatorów przeanalizujmy przykładowy kod. Poniższy listing definiuje funkcję generatora obliczającą serię kwadratów kolejnych liczb całkowitych:

```
>>> def gensquares(N):
    for i in range(N):
        yield i ** 2          # Kolejne wywołanie wznowi wykonanie od tego
                               # miejsca
```

Funkcja zwraca wartość za pomocą instrukcji `yield` i wstrzymuje swoje działanie w pętli `for`; po wznowieniu przywracany jest jej poprzedni stan, w tym ostatnie wartości jego zmiennych `i` oraz `N`, a funkcja kontynuuje swoje działanie od miejsca, w którym została wstrzymana (po instrukcji `yield`). Na przykład, gdy użyjemy tej funkcji w ciele pętli `for`, pierwsza iteracja uruchomi funkcję i pobierze pierwszy wynik; po wznowieniu sterowanie powraca do funkcji po instrukcji `yield` w każdej kolejnej iteracji pętli:

```
>>> for i in gensquares(5):      # Kolejne wznowianie generatora
    print(i, end=' : ')         # Wypisanie ostatniej zwróconej wartości
```

```
0 : 1 : 4 : 9 : 16 :
```

Aby zakończyć generowanie wartości, należy albo jawnie zakończyć funkcję instrukcją `return`, albo pozwolić jej „dobiec do końca” swojego kodu, co również zakończy jej działanie^[1].

Większości użytkowników na pierwszy rzut oka proces ten wydaje się nieco niejasny (jeżeli nie całkowicie magiczny). W rzeczywistości jest to jednak dosyć proste. Jeżeli naprawdę chcesz

zobaczyć, co się dzieje w środku pętli `for`, powinieneś bezpośrednio wywołać funkcję generatora:

```
>>> x = gensquares(4)
>>> x
<generator object gensquares at 0x000000000292CA68>
```

Otrzymujemy obiekt generatora obsługujący protokół iteracji omówiony w rozdziale 14. — funkcja generatora zostaje tak skompilowana, aby automatycznie zwracać taki obiekt. Zwrócony obiekt generatora posiada metodę `__next__`, która uruchamia funkcję lub wznowia jej działanie w miejscu poprzedniego wywołania instrukcji `yield`, a w przypadku zakończenia sekwencji wywołuje wyjątek `StopIteration`. Funkcja wbudowana `next(X)` wywołuje metodę `X.__next__()` obiektu generatora (w wersji 3.x) lub `X.next()` w wersji 2.x:

```
>>> next(x)                                # W wersji 3.x równoważne z x.__next__()
0
>>> next(x)                                # W wersji 2.x można użyć x.next() lub
next(x)
1
>>> next(x)
4
>>> next(x)
9
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Jak wiemy z rozdziału 14., pętle `for` (oraz inne konteksty iteracyjne) działają z generatorami właśnie w taki sposób: wywołują ich metodę `__next__` przy każdej iteracji, aż zostanie wywołany wyjątek. Wynikiem działania generatora jest sekwencja wartości zwracanych kolejno po jednej przy każdej iteracji. Jeżeli iterowany obiekt nie obsługuje tego protokołu, pętla `for` wykorzysta zastępco protokół indeksowania po elementach sekwencji.

Zauważ, że wywołanie metody `iter` najwyższego poziomu protokołu iteracji nie jest tutaj wymagane, ponieważ generatorы są swoimi własnymi iteratorami, obsługującymi tylko jedną, aktywną iterację jednocześnie. Innymi słowy, generatorы nie korzystają z metody `iter`, ponieważ obsługują bezpośrednio metodę `next`. Dotyczy to również wyrażeń generatora, które poznamy w dalszej części tego rozdziału (więcej na ten temat już niebawem):

```
>>> y = gensquares(5)                      # Zwraca generator, który jest swoim
wlasnym iteratorem
>>> iter(y) is y                           # Wywołanie metody iter() nie jest
wymagane
True
>>> next(y)                               # Możemy od razu wywołać metodę
next()
0
```

Dlaczego funkcje generatorów?

Biorąc pod uwagę proste przykłady, których używamy do zilustrowania podstawowych zagadnień, być może zastanawiasz się, dlaczego w ogóle chciałbyś używać generatorów. Przecież w tym przykładzie moglibyśmy po prostu zbudować od razu listę wszystkich uzyskanych wartości:

```
>>> def buildsquares(n):
    res = []
    for i in range(n): res.append(i ** 2)
    return res

>>> for x in buildsquares(5): print(x, end=' : ')
0 : 1 : 4 : 9 : 16 :
```

Do tego typu iteracji moglibyśmy również posłużyć się pętlą `for`, funkcją `map` lub listą składaną.

```
>>> for x in [n ** 2 for n in range(5)]:
    print(x, end=' : ')
0 : 1 : 4 : 9 : 16 :
>>> for x in map(lambda n: n ** 2, range(5)):
    print(x, end=' : ')
0 : 1 : 4 : 9 : 16 :
```

Generatory mogą jednak okazać się znacznie efektywniejsze z punktu widzenia użycia pamięci i wydajności. Pozwalają funkcjom uniknąć wykonywania całej pracy za jednym zamachem, co jest szczególnie korzystne w przypadku, gdy wynik będzie w całości zajmował dużą ilość pamięci lub jego wygenerowanie zajmie dużo czasu. Generatory pozwalają równomiernie rozłożyć czas generowania danych, pozwalając w tym czasie innemu kodowi przetwarzać już wygenerowane dane częściowe.

Co więcej, w przypadku bardziej zaawansowanych zastosowań generatory mogą stanowić prostą alternatywę dla ręcznego zapisywania stanu między iteracjami w obiektach klasy — w przypadku generatorów zmienne dostępne w zasięgach funkcji są automatycznie zapisywane i przywracane^[2]. Obiekty iterowalne implementowane w postaci klas omówimy w szerszym zakresie w części VI.

Funkcje generatora są również znacznie bardziej ogólne, niż mogłoby Ci się do tej pory wydawać. Mogą operować i zwracać dowolne typy obiektów, a ponieważ są *obiektami iterowalnymi*, mogą pojawiać się w dowolnych kontekstach iteracyjnych, opisywanych w rozdziale 14., w tym w wywołaniach funkcji `tuple`, `wyliczeniach` i słownikach składanych:

```
>>> def ups(line):
    for sub in line.split(','):
        yield sub.upper()                                # Generator podciągów znaków

>>> tuple(ups('aaa,bbb,ccc'))                      # Wszystkie konteksty iteracyjne
('AAA', 'BBB', 'CCC')
>>> {i: s for (i, s) in enumerate(ups('aaa,bbb,ccc'))}
```

```
{0: 'AAA', 1: 'BBB', 2: 'CCC'}
```

Za chwilę pokażemy te same mechanizmy dla wyrażeń generatora — narzędzi, które poświęcają elastyczność funkcji na korzyść związków złożen. W dalszej części tego rozdziału zobaczymy również, że generatorы могут czasami powodować, że rzeczy niemożliwe stają się możliwe, wytwarzając elementy zestawów wyników, które byłyby o wiele za duże, aby wygenerować je wszystkie naraz. Najpierw jednak przejrzymy niektóre zaawansowane funkcje generatorów.

Rozszerzony protokół funkcji generatorów — send kontra next

W Pythonie 2.5 do protokołu generatorów wprowadzono nową metodę `send`, która podobnie jak metoda `__next__` powoduje przejście do kolejnego elementu sekwencji, ale dodatkowo pozwala kodowi wyołującemu skomunikować się z generatorem, aby wpływać na jego działanie.

Z technicznego punktu widzenia od tej pory `yield` staje się wyrażeniem zwracającym dane przekazane w metodzie `send`, a nie instrukcją, jak było dotychczas (choć nadal można je wywoływać na oba sposoby — jako `yield X` lub `A = (yield X)`). Wyrażenie musi zostać ujęte w nawiasy, chyba że po prawej stronie instrukcji `yield` występuje tylko jeden argument. Na przykład `X = yield Y`, ale `X = (yield Y) + 42`.

W przypadku użycia tej dodatkowej składni dane są przesyłane do generatora `G` za pomocą wywołania `G.send(dane)`. Kod generatora jest wznowiany, a wyrażenie `yield` zwraca wartość przekazaną jako `dane`. Jeżeli do wznowienia generatora zostanie użyta metoda `G.__next__()` (lub jej odpowiednik `next(G)`) `yield` zwróci `None`. Na przykład:

```
>>> def gen():
        for i in range(10):
            X = yield i
            print(X)

>>> G = gen()
>>> next(G)           # Jako pierwsze wywołanie musi wystąpić next(), co
uruchamia generator
0
>>> G.send(77)        # Wznowienie generatora z przekazaniem wartości
77
1
>>> G.send(88)
88
2
>>> next(G)           # next() oraz X.__next__() przekazują None
None
3
```

Metoda `send` może być użyta na przykład do zaprogramowania generatora, który może zostać zatrzymany przez wysłanie specjalnego kodu lub przeprogramowany przez przekazanie nowej pozycji w sekwencji przetwarzanej przez generator.

Co więcej, w Pythonie 2.5 i nowszych wersjach do generatorów dodano jeszcze metodę `throw(typ)`, która powoduje zgłoszenie przez generator wyjątku klasy `typ` w miejscu ostatniego wywołania `yield`, oraz metodę `close`, która powoduje wywołanie wyjątku

`GeneratorExit` kończącego działanie generatora. Są to zaawansowane możliwości, nad którymi nie będziemy się dłużej zatrzymywać. Więcej informacji na ten temat można znaleźć w standardowym podręczniku Pythona oraz w części VII tej książki.

Jak już wspomniałem, Python 3.x oferuje funkcję wbudowaną `next(G)`, będącą odpowiednikiem wywołania metody `G.__next__()`. Pozostałe metody generatorów, jak `send`, muszą być wywoływanie jako metody generatorów (na przykład `G.send(X)`). Ten pozorny brak spójności wyjaśnia się, gdy weźmiemy pod uwagę, że wspomniane dodatkowe metody funkcjonują wyłącznie w ramach obiektów generatorów, podczas gdy metoda `__next__()` jest elementem ogólnego interfejsu obiektów iterowalnych (zarówno w przypadku wbudowanych typów, jak i klas zdefiniowanych przez użytkownika).

Zauważ też, że Python 3.3 wprowadza rozszerzenie polecenia `yield` z klauzuli `from`, które pozwala generatorom na delegowanie działań do zagnieżdżonych generatorów. Ponieważ jest to dalsze rozszerzenie tego i tak już dość zaawansowanego zagadnienia, omówimy je pokrótko w najbliższej ramce, a teraz przejdziemy tutaj do narzędzia na tyle zbliżonego, że możemy je nazwać bliźniakiem.

Wyrażenia generatorów — obiekty iterowalne spotykają złożenia

Kiedy okazało się, że opóźnione działanie funkcji generatorów jest tak przydatne, zostało szybko rozszerzone na inne narzędzia. Zarówno w wersji 2.x, jak i 3.x Pythona pojęcia obiektów iterowalnych i list składanych zostały uzupełnione o nową cechę języka — *wyrażenia generatorów*. Składniowo wyrażenie generatora wygląda podobnie do zwykłej listy składanej i obsługuje jej pełną składnię, z filtrami `if` i zagnieżdżaniem pętli włącznie, z tym że zamiast nawiasów kwadratowych stosuje się nawiasy okrągłe (które podobnie jak w przypadku krotek często są opcjonalne):

```
>>> [x ** 2 for x in range(4)]          # Lista składana: zwraca listę
[0, 1, 4, 9]
>>> (x ** 2 for x in range(4))        # Wyrażenie generatora: zwraca obiekt
iterowalny
<generator object <genexpr> at 0x00000000029A8288>
```

W rzeczywistości, przynajmniej pod względem funkcjonalności, tworzenie wyrażenia listy składanej jest zasadniczo takie samo jak opakowanie wyrażenia generatora w wywołanie wbudowanej funkcji `list` — zmusza go do wygenerowania wszystkich wyników z listy jednocześnie:

```
>>> list(x ** 2 for x in range(4))      # Odpowiednik listy składanej
[0, 1, 4, 9]
```

Z operacyjnego punktu widzenia wyrażenia generatora bardzo się jednak różnią: zamiast budować listę wyników w pamięci, zwracając obiekt *generatora* — automatycznie tworzony obiekt iterowalny, który obsługuje protokół *iteracji*, pozwalający na uzyskiwanie po jednym elemencie listy wyników w dowolnym kontekście iteracji. Obiekt iterowalny zachowuje również stan generatora, gdy jest aktywny (przykładem może być zmienna `x` w poprzednich wyrażeniach) wraz z lokalizacją kodu generatora.

Efekt jest bardzo podobny do działania funkcji generatora, ale w kontekście *wyrażenia składanego*: otrzymujemy obiekt, który pamięta, gdzie zostało przerwane jego działanie po zwróceniu każdej kolejnej części wyniku. Podobnie jak to miało miejsce w przypadku funkcji generatora, spojrzenie „pod maskę” na protokół iteracji automatycznie obsługiwany przez te obiekty może pomóc w odkryciu ich sposobu działania; i podobnie jak poprzednio, wywołanie

funkcji `iter` na najwyższym poziomie nie jest tutaj wymagane z powodów, które przedstawimy już niebawem:

```
>>> G = (x ** 2 for x in range(4))
>>> iter(G) is G                                # iter(G) jest opcjonalne: __iter__
zwraca self
True
>>> next(G)                                     # Obiekty generatorów: metoda
automatyczna
0
>>> next(G)
1
>>> next(G)
4
>>> next(G)
9
>>> next(G)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
>>> G
<generator object <genexpr> at 0x00000000029A8318>
```

Jednak najczęściej nie mamy okazji ręcznie używać kolejnych wywołań metody `next` iteratora, ponieważ w kontekście pętli `for` odbywa się to automatycznie:

```
>>> for num in (x ** 2 for x in range(4)):          # Automatyczne wywołanie
metody next()
    print('%s, %s' % (num, num / 2.0))

0, 0.0
1, 0.5
4, 2.0
9, 4.5
```

Jak już mieliśmy okazję zaobserwować, każdy kontekst iteracyjny działa w taki sposób: pętle `for`, funkcje wbudowane `sum`, `map` i `sorted`, listy składane i inne konteksty iteracyjne, które omawialiśmy w rozdziale 14., takie jak funkcje wbudowane `any`, `all` czy `list`. Będąc *obiektami iterowalnymi*, wyrażenia generatora mogą pojawiać się w dowolnym z tych kontekstów iteracji, podobnie jak wyniki wywołania funkcji generatora.

Przykład kodu przedstawiony poniżej wdraża wyrażenia generatora w wywołaniu metody `join` ciągów znaków i przypisaniu krotek. W pierwszym przykładzie metoda `join` uruchamia generator i łączy wytwarzane podciągi bez żadnych separatorów, po prostu aby je połączyć:

```
>>> ''.join(x.upper() for x in 'aaa,bbb,ccc'.split(','))
```

```
'AAABBBCCC'  
>>> a, b, c = (x + '\n' for x in 'aaa,bbb,ccc'.split(','))  
>>> a, c  
('aaa\n', 'ccc\n')
```

Zwróć uwagę, że wywołanie metody `join` w powyższym przykładzie nie wymaga użycia dodatkowych nawiasów wokół generatora. Z punktu widzenia składni nawiasy *nie są wymagane* wokół wyrażenia generatora, które jest jedynym elementem już zawartym w nawiasach używanych do innych celów — takich jak wywołanie funkcji. Nawiasy są wymagane we wszystkich innych przypadkach, nawet jeżeli wydają się nadmiarowe, tak jak to pokazano w drugim przykładzie z wywołaniem funkcji `sorted` poniżej:

```
>>> sum(x ** 2 for x in range(4)) # Tutaj nawiasy są opcjonalne  
14  
>>> sorted(x ** 2 for x in range(4)) # Tutaj nawiasy są opcjonalne  
[0, 1, 4, 9]  
>>> sorted((x ** 2 for x in range(4)), reverse=True) # Tutaj nawiasy są wymagane!  
[9, 4, 1, 0]
```

Podobnie jak w przypadku opcjonalnych nawiasów w krotkach, nie ma tu żadnej formalnej reguły opisującej stosowanie nawiasów w tym zakresie, chociaż wyrażenia generatora nie mają tak wyraźnie określonej roli, jak stała kolekcja innych obiektów, takich jak krotki, co sprawia, że w tym kontekście dodatkowe nawiasy częściej wydają się być nadmiarowe.

Dlaczego wyrażenia generatora?

Podobnie jak funkcje generatora, wyrażenia generatora są optymalizacją *zajętości pamięci* — nie wymagają, aby cała lista wyników była tworzona naraz, jak robi to lista w nawiasach kwadratowych. Podobnie jak funkcje generatora, wyrażenia dzielą tworzenie wyników na mniejsze przedziały czasowe — dają wyniki po jednym na raz, zamiast zmuszać kod wywołujący do oczekiwania na utworzenie pełnego zestawu w jednym wywołaniu.

Z drugiej strony wyrażenia generatora w praktyce mogą działać nieco *wolniej* niż listy składane, więc prawdopodobnie najlepiej nadają się do przetwarzania bardzo dużych zestawów danych lub aplikacji, które nie mogą czekać na wygenerowanie pełnych wyników. Bardziej autorytatywne opinie dotyczące wydajności będą jednak musiały poczekać aż do omówienia skryptów pomiaru czasu, którymi będziemy zajmować się w następnym rozdziale.

Choć to moja nieco subiektywna opinia, to jednak warto zauważyć, że wyrażenia generatora ujawniają swoje zalety również podczas kodowania czy — jak kto woli — tworzenia kodu programu, o czym opowiemy już w kolejnych sekcjach.

Wyrażenia generatora a funkcja map

Jednym ze sposobów przekonania się o zaletach kodowania wyrażeń generatora jest porównanie ich z innymi narzędziami funkcyjnymi, tak jak zrobiliśmy to w przypadku list składanych. Na przykład wyrażenia generatora często są równoważne wywołaniom funkcji `map` w wersji 3.x, ponieważ oba rozwiązania generują wyniki na żądanie. Podobnie jak w przypadku list składanych, wyrażenia generatora mogą być łatwiejsze do zakodowania, gdy zastosowana operacja nie jest wywołaniem funkcji. W wersji 2.x funkcja `map` tworzy tymczasowe listy, ale

wyrażenia generatora już tak nie robią, choć nadal obowiązują tutaj te same porównania sposobów kodowania:

```
>>> list(map(abs, (-1, -2, 3, 4)))          # Funkcja map wywołana
na krotce
[1, 2, 3, 4]
>>> list(abs(x) for x in (-1, -2, 3, 4))      # Wyrażenie generatora
[1, 2, 3, 4]
>>> list(map(lambda x: x * 2, (1, 2, 3, 4)))    # Przypadek wywołania
bez funkcji (z wyrażeniem lambda)
[2, 4, 6, 8]
>>> list(x * 2 for x in (1, 2, 3, 4))          # Prostsze niż
generator?
[2, 4, 6, 8]
```

To samo dotyczy operacji przetwarzania tekstów, takich jak wywołanie metody `join`, które widzieliśmy wcześniej — listy składane tworzą dodatkową, tymczasową listę wyników, co jest całkowicie *bezelowe* w tym kontekście, ponieważ lista nie jest zachowywana, a użycie funkcji `map` traci swoją prostotę w porównaniu ze składnią wyrażenia generatora, gdy odwzorowywana operacja nie jest wywołaniem innej funkcji:

```
>>> line = 'aaa,bbb,ccc'
>>> ''.join([x.upper() for x in line.split(',')])      # Tworzy
bezelową listę
'AAABBBCCC'
>>> ''.join(x.upper() for x in line.split(','))        # Generuje
wyniki
'AAABBBCCC'
>>> ''.join(map(str.upper, line.split(',')))           # Generuje
wyniki
'AAABBBCCC'
>>> ''.join(x * 2 for x in line.split(','))            # Prostsze niż
generator?
'aaaaaaaaaaaaaaaaaaaaaa'
>>> ''.join(map(lambda x: x * 2, line.split(',')))
'aaaaaaaaaaaaaaaaaaaaaa'
```

Zarówno funkcja `map`, jak i wyrażenia generatora mogą być dowolnie zagnieżdżone, co zapewnia wygodne używanie ich w programach i wymaga wywołania funkcji `list` lub innego kontekstu iteracji do rozpoczęcia procesu generowania wyników. Na przykład lista składana w przykładzie poniżej daje taki sam rezultat jak wywołanie funkcji `map` w wersji 3.x i równoważne generatory, które po niej następują, ale tworzy dwie listy fizyczne; inne rozwiązania generują tylko jedną liczbę całkowitą na raz za pomocą zagnieżdżonych generatorów, a same wyrażenia generatora mogą wyraźniej odzwierciedlać jego przeznaczenie:

```
>>> [x * 2 for x in [abs(x) for x in (-1, -2, 3, 4)]]      # Zagnieżdżone
listy składane
```

```
[2, 4, 6, 8]
>>> list(map(lambda x: x * 2, map(abs, (-1, -2, 3, 4))))      # Zagnieżdżone
funkcje map
[2, 4, 6, 8]
>>> list(x * 2 for x in (abs(x) for x in (-1, -2, 3, 4)))      # Zagnieżdżone
generatory
[2, 4, 6, 8]
```

Chociaż efektem wszystkich trzech przykładów jest łączenie operacji, generatory robią to bez tworzenia wielu list tymczasowych. W kolejnym przykładzie dla wersji 3.x jednocześnie zagnieżdżamy i łączymy generatory — wyrażenie zagnieżdżonego generatora jest aktywowane przez funkcję `map`, która z kolei jest aktywowana przez wywołanie funkcji `list`.

```
>>> import math
>>> list(map(math.sqrt, (x ** 2 for x in range(4))))      #
Zagnieżdżone kombinacje
[0.0, 1.0, 2.0, 3.0]
```

Technicznie rzecz biorąc, `range` po prawej stronie wyrażenia w przykładzie powyżej w wersji 3.x jest również generatorem wartości, aktywowanym przez samo wyrażenie generatora — mamy tutaj *trzy poziomy* generowania wartości, które wytwarzają indywidualne wartości od wewnętrz do zewnętrz tylko na żądanie; takie rozwiązywanie działa ze względu na specyficzne narzędzia iteracyjne i protokół iteracji Pythona. W rzeczywistości zagnieżdżenia generatorów mogą być dowolnie mieszane i głębokie, chociaż niektóre mogą być „bardziej słuszne” niż inne:

```
>>> list(map(abs, map(abs, map(abs, (-1, 0, 1)))))      # Zagnieżdżanie poszło
nie tak, jak powinno?
[1, 0, 1]
>>> list(abs(x) for x in (abs(x) for x in (abs(x) for x in (-1, 0, 1))))
[1, 0, 1]
```

Te ostatnie przykłady ilustrują, jak ogólne mogą być generatory, ale jednocześnie zostały również zakodowane w celowo złożonej formie, aby podkreślić, że wyrażenia generatora mają taki sam potencjał do nadużyć jak omówione wcześniej listy złożone — jak zwykle powinieneś zachować prostotę swoich rozwiązań, chyba że z takich czy innych powodów muszą pozostać złożone — to motyw, do którego powróćmy w dalszej części tego rozdziału.

Jeżeli jednak są dobrze używane, wyrażenia generatora łączą wyrazistość wyrażeń list złożonych z korzyściami innych iteracji. Tutaj na przykład podejście bez zagnieżdżania zapewniają prostsze rozwiązyania, ale nadal wykorzystują mocne strony generatorów — zgodnie z motto Pythona płaska struktura jest zazwyczaj lepsza niż zagnieżdżona:

```
>>> list(abs(x) * 2 for x in (-1, -2, 3, 4))      # Płaski
(niezagnieżdżony) odpowiednik
[2, 4, 6, 8]
>>> list(math.sqrt(x ** 2) for x in range(4))      # Płaska struktura
jest często lepsza
[0.0, 1.0, 2.0, 3.0]
>>> list(abs(x) for x in (-1, 0, 1))
[1, 0, 1]
```

Wyrażenia generatora a filtry

Wyrażenia generatora obsługują również całą standardową składnię list składanych — w tym klauzule `if` działające jak klauzule `filter`, które spotkaliśmy już wcześniej. Ponieważ klauzula `filter` jest iterowalna w wersji 3.x, która generuje wyniki na żądanie, wyrażenie generatora z klauzulą `if` jest operacyjnie równoważne (w wersji 2.x klauzula `filter` tworzy listę tymczasową, której nie robi generator, ale podobnie jak poprzednio, kod możemy porównywać w podobny sposób). W przykładzie pokazanym poniżej wywołanie metody `join` wystarczy, aby zmusić oba rozwiązania do wygenerowania wyników:

```
>>> line = 'aa bbb c'  
>>> ''.join(x for x in line.split() if len(x) > 1) # Generator z  
klauzulą 'if'  
'aabbb'  
>>> ''.join(filter(lambda x: len(x) > 1, line.split())) # Podobne do  
klauzuli filter  
'aabbb'
```

Generator wydaje się tutaj nieco prostszy niż klauzula `filter`. Jednak w przypadku list składanych dodawanie kolejnych kroków przetwarzania do klauzuli `filter` wyników wymaga również zastosowania funkcji `map`, co czyni klauzulę `filter` znacznie bardziej złożoną niż wyrażenie generatora:

```
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)  
'AABBB'  
>>> ''.join(map(str.upper, filter(lambda x: len(x) > 1, line.split())))  
'AABBB'
```

W efekcie wyrażenia generatora wykonują dla obiektów iterowalnych w wersji 3.x, takich jak `map` i `filter`, to, co listy składane robią dla tworzenia list w wywołaniach w wersji 2.x — zapewniają więcej ogólnych struktur kodowania, które nie zależą od funkcji, ale nadal opóźniają generowanie wyników. Podobnie jak w przypadku list, zawsze istnieje odpowiednik wyrażenia generatora oparty na instrukcjach, chociaż czasami wymaga zastosowania znacznie większej ilości kodu:

```
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)  
'AABBB'  
>>> res = ''  
>>> for x in line.split(): # Odpowiednik wyrażenia  
generatora?  
    if len(x) > 1: # To także odpowiednik  
        join  
        res += x.upper()  
>>> res  
'AABBB'
```

W tym przypadku jednak forma oparta na instrukcjach nie jest taka sama — nie może wytwarzać elementów pojedynczo, a także emuluje efekt wywołania metody `join`, który wymusza generowanie wszystkich wyników naraz. Prawdziwym odpowiednikiem wyrażenia generatora byłaby funkcja generatora z instrukcją `yield`, jak to pokażemy w następnej sekcji.

Funkcje generatorów a wyrażenia generatorów

Podsumujmy to, co udało nam się do tej pory omówić w tej sekcji:

Funkcje generatora

Definicja funkcji z instrukcją `def` zawierającą instrukcję `yield` jest przekształcana w funkcję generatora. Po wywołaniu zwraca nowy *obiekt generatora* z automatycznym zachowaniem lokalnego zasięgu i pozycji kodu, automatycznie utworzoną metodą `__iter__`, która zwraca sam obiekt, oraz automatycznie utworzoną metodą `__next__` (`next` w wersji 2.x), która uruchamia funkcję lub wznowia ją w miejscu, w którym została ostatnio przerwana, i zgłasza wyjątek `StopIteration` po zakończeniu tworzenia wyników.

Wyrażenia generatora

Wyrażenie składane umieszczone w nawiasach znane jest jako wyrażenie generatora. Po uruchomieniu zwraca nowy *obiekt generatora* z tym samym automatycznie utworzonym interfejsem metody i zachowaniem stanu jak w wynikach wywołania funkcji generatora — z metodą `__iter__`, która po prostu zwraca siebie, oraz metodą `__next__` (`next` w wersji 2.x), która uruchamia domyślną pętlę lub wznowia ją w miejscu, w którym została przerwana, i zgłasza wyjątek `StopIteration` po zakończeniu tworzenia wyników.

Efektem działania jest generowanie wyników na żądanie w kontekstach iteracyjnych, w których te interfejsy są stosowane automatycznie.

Co interesujące, często tę samą iterację można zapisać *zarówno* w postaci funkcji generatora jak i wyrażenia generatora. Poniższy przykład tworzy wyrażenie generatora, powtarzające czterokrotnie każdą literę podanego ciągu znaków:

```
>>> G = (c * 4 for c in 'JAJKO')          # Wyrażenie generatora
>>> list(G)                                # Wymuszenie wygenerowania wyników
w całości
['AAAA', 'JJJJ', 'KKKK', '0000']
```

Równoważna funkcja generatora wymaga nieco więcej kodu, ale jako funkcja składająca się z wielu poleceń będzie mogła zawierać więcej logiki i w razie potrzeby wykorzystywać więcej informacji o stanie. W rzeczywistości jest to w zasadzie to samo co kompromis pomiędzy wyrażeniami `lambda` a funkcjami `def` — związek wyrażenia kontra wielkie możliwości rozbudowanego kodu:

```
>>> def timesfour(S):                      # Funkcja generatora
    for c in S:
        yield c * 4
>>> G = timesfour('jajko')
>>> list(G)                                # Iteracja automatyczna
['aaaa', 'jjjj', 'kkkk', 'oooo']
```

Dla większości użytkowników oba rozwiązania są bardziej do siebie podobne niż różne. Zarówno wyrażenia, jak i funkcje obsługują iterację automatyczną i ręczną. W powyższym liście wywołanie funkcji `list` wykorzystuje iterację automatyczną, poniżej przedstawiamy przykład iteracji ręcznej.

```
>>> G = (c * 4 for c in 'JAJKO')
>>> I = iter(G)                            # Ręczna iteracja (wyrażenie)
>>> next(I)
```

```
'JJJJ'

>>> next(I)
'AAAA'

>>> G = timesfour('jajko')
>>> I = iter(G)      )
# Ręczna iteracja (funkcja)

>>> next(I)
'jjjj'
>>> next(I)
'aaaa'
```

W obu przypadkach Python automatycznie tworzy obiekt generatora, który posiada zarówno metody wymagane przez protokół iteracji, jak i mechanizmy przechowywania stanu dla zmiennych w kodzie generatora i jego bieżącej lokalizacji. Zwróć uwagę, w jaki sposób tworzymy tutaj nowe generatory do iteracji — jak pokażemy w następnej sekcji, generatory są iteratorami jednorazowymi.

Najpierw jednak pokażemy oparty na instrukcjach odpowiednik wyrażenia znajdującego się na końcu poprzedniej sekcji: funkcję zwracającą wartości — choć tak naprawdę różnica nie ma znaczenia, jeżeli kod, który jej używa, generuje wszystkie wyniki za pomocą narzędzi takiego jak metoda `join`:

```
>>> line = 'aa bbb c'
>>> ''.join(x.upper() for x in line.split() if len(x) > 1)      # Wyrażenie
'AABBB'

>>> def gensub(line):                                              # Funkcja
    for x in line.split():
        if len(x) > 1:
            yield x.upper()

>>> ''.join(gensub(line))                                         # Ale po co
nam tutaj generator?
'AABBB'
```

Chociaż generatory pełnią ważne role, w takich przypadkach użycie generatorów w stosunku do pokazanego wcześniej prostego kodu może być trudne do uzasadnienia, z wyjątkiem powodów stylistycznych. Z drugiej strony wymiana czterech wierszy kodu na jeden może wielu użytkownikom wydawać się dość atrakcyjnym rozwiązaniem!

Generatory są obiektami o jednoprzebiegowej iteracji

Subtelny, ale ważny punkt: zarówno funkcje generatora, jak i wyrażenia generatora są swoimi własnymi iteratorami, a zatem obsługują tylko *jedną aktywną iterację* — i w przeciwieństwie do niektórych typów wbudowanych nie mogą mieć wielu iteratorów działających w różnych lokalizacjach swojego zestawu wyników. Z tego powodu iteratorem generatora jest sam generator; w rzeczywistości, jak zasugerowano wcześniej, wywołanie metody `iter` na wyrażeniu lub funkcji generatora jest całkowicie opcjonalne i nie przynosi żadnych rezultatów:

```
>>> G = (c * 4 for c in 'JAJKO')
```

```
>>> iter(G) is G          # Iterator generatora jest tym samym
generatorem: G posiada metodę __next__
True
```

Jeżeli za pomocą kilku iteratorów spróbujemy ręcznie iterować strumień wyników z jednego generatora, zauważmy, że w danym momencie wszystkie iteratory znajdą się na tej samej pozycji.

```
>>> G = (c * 4 for c in 'JAJKO')  # Utworzenie nowego generatora
>>> I1 = iter(G)                  # Ręczna iteracja
>>> next(I1)
'JJJJ'
>>> next(I1)
'AAAA'
>>> I2 = iter(G)                  # Drugi iterator znajduje się na tej samej
pozycji!
>>> next(I2)
'JJJJ'
```

Co więcej, gdy jedna iteracja dojdzie do końca sekwencji, to samo dzieje się z pozostałymi. Aby zacząć od nowa, należy utworzyć nowy generator:

```
>>> list(I1)                   # Odczyt pozostałych elementów z I1
['0000']
>>> next(I2)                   # Pozostałe iteratory są również opróżnione
StopIteration
>>> I3 = iter(G)                # Podobnie nowo tworzone iteratory
>>> next(I3)
StopIteration
>>> I3 = iter(c * 4 for c in 'JAJKO') # Nowy generator zaczyna od początku
>>> next(I3)
'JJJJ'
```

Taka sama sytuacja ma miejsce w przypadku funkcji generatorów — poniższy przykład prezentuje funkcję generatora, której iterator wyczerpuje się po pierwszym przejściu:

```
>>> def timesfour(S):
    for c in S:
        yield c * 4
>>> G = timesfour('jajko')      # Funkcje generatorów działają w ten sam
sposób
>>> iter(G) is G
True
```

```

>>> I1, I2 = iter(G), iter(G)
>>> next(I1)
'jjjj'
>>> next(I1)
'aaaa'
>>> next(I2)                                # I2 znajduje się na tej samej pozycji co I1
'jjjj'

```

To znacząca różnica w porównaniu z niektórymi typami wbudowanymi, które obsługują wielokrotne iteracje i przejścia, a zmiany w ich wartości są odzwierciedlane w aktywnych iteratorach:

```

>>> L = [1, 2, 3, 4]
>>> I1, I2 = iter(L), iter(L)
>>> next(I1)
1
>>> next(I1)
2
>>> next(I2)                                # Listy obsługują wielokrotną iterację
1
>>> del L[2:]                                # Zmiany w liście są odzwierciedlane w
iteratorach
>>> next(I1)
StopIteration

```

Chociaż nie jest to oczywiste w tych prostych przykładach, może mieć znaczenie w kodzie: jeżeli chcesz skanować wartości generatora wiele razy, musisz albo utworzyć nowy generator dla każdego przebiegu, albo zbudować listę wygenerowanych wartości, której możesz później wielokrotnie używać — wartości z pojedynczego generatora ulegają wyczerpaniu po jednym przebiegu całego cyklu. Doskonały przykład kodu, który musi uwzględnić tę właściwość generatorów, znajdziesz w ramce „Warto pamiętać: iteratory jednoprzebiegowe”.

Kiedy w części VI zaczniemy tworzyć własne obiekty iterowalne oparte na klasach, przekonasz się również, że to od Ciebie będzie zależało, ile iteracji będzie obsługiwał dany obiekt (o ile w ogóle). Ogólnie rzecz biorąc, obiekty, które chcą obsługiwać wiele niezależnych skanów iteracji, zwracają dodatkowe instancje obiektów tej klasy zamiast samych siebie. W następnej sekcji znajdziesz przegląd tego modelu.

Rozszerzona składnia instrukcji yield w Pythonie 3.3

W Pythonie 3.3 wprowadzono rozszerzoną składnię instrukcji `yield`, która za pomocą klauzuli `from generator` pozwala na delegowanie działania do subgeneratora. W prostych przypadkach jest to odpowiednik pętli `for` — w poniższym przykładzie wywołanie funkcji `list` zmusza generator do wygenerowania od razu wszystkich swoich wartości, a lista składana w nawiasach to wyrażenie generatora, o którym pisaliśmy wcześniej w tym rozdziale:

```
>>> def both(N):
```

```

        for i in range(N): yield i
        for i in (x ** 2 for x in range(N)): yield i
>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]

```

Nowa składnia w wersji 3.3 sprawia, że jest to prawdopodobnie bardziej zwięzłe i wyraźne oraz obsługuje wszystkie zwykłe konteksty użycia generatora:

```

>>> def both(N):
        yield from range(N)
        yield from (x ** 2 for x in range(N))
>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]
>>> ' : '.join(str(i) for i in both(5))
'0 : 1 : 2 : 3 : 4 : 0 : 1 : 4 : 9 : 16'

```

Jednak w bardziej zaawansowanych zastosowaniach takie rozszerzenie pozwala subgeneratorom odbierać wartości send i throw bezpośrednio z zasięgu wywoływania i zwracać ostateczną wartość do zewnętrznego generatora. W efekcie możliwe staje się podzielenie takich generatorów na wiele subgeneratorów, podobnie jak jedną funkcję można podzielić na wiele podfunkcji.

Ponieważ takie możliwości są dostępne tylko w wersji 3.3 i nowszych oraz wykraczają daleko poza zakres tematyki tego rozdziału, nie będziemy się nimi tutaj więcej zajmować. Więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji Pythona 3.3 i nowszych wersji. Dodatkowy przykład zastosowania polecenia `yield from` znajdziesz w rozwiążaniu ćwiczenia 11. do tej części książki, opisanego na końcu rozdziału 21.

Generowanie wyników we wbudowanych typach, narzędziach i klasach

Choć w tej sekcji skupiliśmy się głównie na tworzeniu generatorów wartości, powinieneś pamiętać, że wiele typów wbudowanych również zachowuje się w taki sposób. Na przykład, jak widzieliśmy w rozdziale 14., słowniki są obiektami iterowalnymi posiadającymi iteratory generujące klucze przy każdej iteracji:

```

>>> D = {'a':1, 'b':2, 'c':3}
>>> x = iter(D)
>>> next(x)
'a'
>>> next(x)
'c'

```

Klucze słowników, podobnie jak generatory tworzone przez użytkownika, mogą być iterowane wielokrotnie, zarówno ręcznie, jak i w wielopoziomowych kontekstach iteracyjnych, takich jak pętle `for`, wywołania `map`, listy składane i wiele innych, które poznaliśmy w rozdziale 14.

```
>>> for key in D:  
    print(key, D[key])  
  
c 3  
b 2  
a 1
```

Jak również mieliśmy okazję zaobserwować, w przypadku iteratorów plików Python po prostu wczytuje kolejne wiersze na żądanie:

```
>>> for line in open('temp.txt'):  
    print(line, end=' ')  
  
To tylko  
rana powierzchnia.
```

Wbudowane iteratory typów są związane z konkretnym typem wyników, ale cała koncepcja jest zbliżona do tego, co zaprogramowaliśmy z użyciem wyrażeń i funkcji. Konteksty iteracyjne, jak pętle, akceptują dowolny obiekt iterowany, czy to zdefiniowany przez użytkownika, czy też wbudowany.

Generatory i narzędzia biblioteczne: skanery katalogów

Choć takie zagadnienia wykraczają nieco poza zakres tej książki, warto wspomnieć, że wiele standardowych narzędzi bibliotecznych Pythona również generuje wartości; są to m.in. parsery poczty e-mail czy standardowy moduł *skanujący katalogi* (ang. *directory walker*), który dla każdego poziomu drzewa katalogów tworzy krotkę bieżącego katalogu, jego podkatalogów i plików:

```
>>> import os  
  
>>> for (root, subs, files) in os.walk('.'):           # Generator  
    skanujący katalogi  
  
        for name in files:                            # Pythonowy  
    odpowiednik operacji 'find'  
  
            if name.startswith('call'):  
  
                print(root, name)  
  
. callables.py  
. \dualpkg callables.py
```

W rzeczywistości funkcja `os.walk` jest kodowana jako funkcja rekurencyjna w standardowym pliku biblioteki `os.py`, znajdującym się w katalogu np. `C:\Python33\Lib` w systemie Windows. Ponieważ do zwracania wyników używa polecenia `yield` (a w wersji 3.3 `yield from` zamiast pętli `for`), jest to normalna funkcja generatora, a zatem obiekt iterowalny:

```
>>> G = os.walk(r'C:\code\pkg')  
  
>>> iter(G) is G                      # Iterator jednoprzepięciowy;  
wywołanie iter(G) jest opcjonalne  
  
True  
  
>>> I = iter(G)  
  
>>> next(I)
```

```

('C:\\\\code\\\\pkg', ['__pycache__', ['eggs.py', 'eggs.pyc', 'main.py',
...etc...])
>>> next(I)
('C:\\\\code\\\\pkg\\\\__pycache__', [], ['eggs.cpython-33.pyc', ...etc...])
>>> next(I)
StopIteration

```

Dzięki podawaniu wyników na bieżąco funkcja skanująca katalogi nie zmusza swoich klientów do oczekiwania na przeskanowanie całego drzewa. Więcej informacji na temat tego narzędzia można znaleźć w podręcznikach Pythona i wielu innych książkach. Zobacz także rozdział 14. i inne, w których używamy funkcji `os.popen` — jest to pokrewny obiekt iterowalny używany do uruchamiania polecenia powłoki i odczytywania jego rezultatów.

Generatory i funkcje aplikacji

W rozdziale 18. zauważliśmy, że argumenty oznaczone gwiazdką mogą rozpakować obiekty iterowalne do postaci poszczególnych argumentów. Teraz gdy widzieliśmy generatorów, możemy się również przekonać, co to oznacza w kodzie. Przedstawiony przykład działa zarówno w wersji 3.x, jak i 2.x (choćżeż `range` w wersji 2.x to lista):

```

>>> def f(a, b, c): print('%s, %s, and %s' % (a, b, c))
>>> f(0, 1, 2)                                     # Normalne argumenty pozycyjne
0, 1, and 2
>>> f(*range(3))                                # Rozpakowanie wartości range: w
wersji 3.x to obiekt iterowalny
0, 1, and 2
>>> f(*(i for i in range(3)))                  # Rozpakowanie wartości wyrażenia
generatora
0, 1 i 2

```

Dotyczy to także słowników i widoków (choćżeż `dict.values` to także lista w wersji 2.x, a kolejność elementów jest dowolna przy przekazywaniu wartości według pozycji):

```

>>> D = dict(a='Bob', b='dev', c=40.5); D
{'b': 'dev', 'c': 40.5, 'a': 'Bob'}
>>> f(a='Bob', b='dev', c=40.5)            # Normalne argumenty ze słowami
kluczowymi
Bob, dev, and 40.5
>>> f(**D)                                    # Rozpakowanie słownika: klucz=wartość
Bob, dev, and 40.5
>>> f(*D)                                     # Rozpakowanie iteratora kluczy
b, c, and a
>>> f(*D.values())                           # Rozpakowanie iteratora widoku: w
wersji 3.x to obiekt iterowalny
dev, 40.5, and Bob

```

Ponieważ wbudowana funkcja `print` w wersji 3.x wypisuje wszystkie swoje argumenty, sprawia to, że następujące trzy formy są równoważne — ta ostatnia używa `*` do rozpakowania wyników wymuszonych z wyrażenia generatora (druga forma tworzy również listę wartości zwracanych, a pierwsza może pozostawić cursor na końcu wiersza wyjściowego w niektórych powłokach, ale nie w GUI IDLE):

```
>>> for x in 'spam': print(x.upper(), end=' ')
S P A M
>>> list(print(x.upper(), end=' ') for x in 'spam')
S P A M [None, None, None, None]
>>> print(*(x.upper() for x in 'spam'))
S P A M
```

Dodatkowy przykład, który używa iteratora do rozpakowania wierszy pliku do postaci argumentów, znajduje się w rozdziale 14.

Przegląd: obiekty iterowalne definiowane przez użytkownika w klasach

Chociaż znów wykracza to poza zakres tego rozdziału, warto dodać, że możliwe jest również implementowanie dowolnych obiektów generatora zdefiniowanych przez użytkownika za pomocą klas zgodnych z protokołem iteracji. Takie klasy muszą definiować specjalną metodę `__iter__` wywoływaną przez wbudowaną funkcję `iter`, która musi zwracać obiekt posiadający metodę `__next__` (`next` w wersji 2.x) uruchamianą przez wbudowaną funkcję `next`:

```
class SomeIterable:
    def __iter__(...): ... # Po wywołaniu iter(): zwraca self
    lub obiekt pomocniczy
    def __next__(...): ... # Po wywołaniu next(): zdefiniowane
    tutaj lub w innej klasie
```

Jak sugerowano w poprzedniej sekcji, klasy te zwykle zwracają swoje obiekty bezpośrednio w trybie iteracji jednoprzebiegowej lub zwracają obiekty dodatkowe ze stanem specyficzny dla danego skanu przy obsłudze iteracji wieloprzebiegowej.

Alternatywnie zdefiniowane przez użytkownika metody klas iterowalnych mogą czasami wykorzystywać polecenie `yield` do przekształcania się w generator, zautomatycznie utworzoną metodą `__next__` — jest to dosyć powszechnie zastosowanie polecenia `yield`, szerzej opisywane w rozdziale 30., które jest mocno niejawne, ale potencjalnie bardzo użyteczne! Metoda indeksowania `__getitem__` jest również dostępna jako opcja rezerwowa dla iteracji, chociaż często nie jest tak elastyczna jak schemat `__iter__ i __next__` (ale ma swoje zalety w zakresie kodowania sekwencji).

Obiekty instancji utworzone z takiej klasy są uznawane za iterowalne i mogą być używane w pętlach `for` i wszystkich innych kontekstach iteracji. Jednak dzięki klasom mamy dostęp do bogatszej grupy narzędzi, pozwalającej na definiowanie zaawansowanej logiki i struktur danych, takich jak dziedziczenie, których inne konstrukcje generatorów nie mogą same zaoferować. Dzięki swoim metodom klasy mogą również sprawić, że zachowanie iteracyjne będzie o wiele bardziej *jawne niż „magicznych”* obiektów generatora związanych z wbudowanymi typami oraz funkcjami i wyrażeniami generatora (choć klasa również posiada swoją własną „magię”).

Z tych właśnie względów historia iteratorów i generatorów nie będzie naprawdę kompletna, dopóki nie zobaczymy, w jaki sposób jest ona odwzorowywana na klasy. Na razie będziemy

musieli jednak odłożyć te zagadnienia aż do rozdziału 30., w którym będziemy zajmować się obiektami iterowalnymi opartymi na klasach.

Przykład — generowanie mieszanych sekwencji

Aby zademonstrować moc narzędzi iteracyjnych w działaniu, przejdźmy do bardziej konkretnych przykładów użycia. W rozdziale 18. napisaliśmy funkcję testową, która mieszała kolejność argumentów używanych do testowania uogólnionych funkcji zwracających część wspólną i sumę zbiorów. Zamieściliśmy tam uwagę, że można to lepiej zakodować, korzystając z generatora wartości. Teraz gdy nauczyliśmy się tworzyć generatorы, wykorzystamy ten przykład do praktycznego zastosowania.

Jeszcze jedna uwaga, zanim zaczniemy: ponieważ korzystamy tutaj z wycinania i łączenia obiektów, wszystkie przykłady w tej sekcji (włącznie z permutacjami na końcu) działają tylko na sekwencjach takich jak ciągi znaków i listy, a nie na dowolnych *obiektach iterowalnych*, takich jak pliki, odwzorowania i inne generatorы. Co prawda w niektórych z tych przykładów będą używane generatorы, dostarczające wartości na żądanie, ale nie będziemy przetwarzarzać generatorów jako danych wejściowych. Dostosowanie kodu przykładów dla szerszych kategorii pozostaje kwestią otwartą, choć przedstawiony poniżej kod powinien być wystarczający, jeżeli opakujemy generatorы niesekwencyjne w wywołaniach funkcji `list` przed przekazaniem ich jako argumentów.

Sekwencje mieszające

Jak pokazywaliśmy w rozdziale 18., możemy zmienić kolejność sekwencji z wycinaniem i konkatenacją, przesuwając pierwszy element na koniec w każdej iteracji pętli; *wycinanie* zamiast indeksowania elementu pozwala operatorowi `+` pracować z dowolnymi typami sekwencji:

```
>>> L, S = [1, 2, 3], 'spam'  
>>> for i in range(len(S)):  
    3  
        S = S[1:] + S[:1]                      # Odliczamy w pętli od 0 do  
                                         # Przenosimy pierwszy  
                                         # element na koniec  
        print(S, end=' ')  
  
pams amsp mspa spam  
>>> for i in range(len(L)):  
    L = L[1:] + L[:1]                      # Wycinki działają z  
                                         # dowolnymi typami sekwencji  
    print(L, end=' ')  
[2, 3, 1] [3, 1, 2] [1, 2, 3]
```

W alternatywny sposób, jak to widzieliśmy w rozdziale 13., możemy uzyskać te same wyniki, przesuwając całą sekcję z przodu na koniec sekwencji, chociaż w takiej sytuacji kolejność wyników nieznacznie się różni:

```
>>> for i in range(len(S)):  
    X = S[i:] + S[:i]                      # Dla pozycji 0..3  
                                         # Część końcowa + część  
                                         # początkowa (ten sam efekt)  
    print(X, end=' ')
```

```
spam pams amsp mspa
```

Proste funkcje

W przedstawionej postaci nasz kod działa poprawnie tylko na zmiennych o określonych nazwach. Aby go uogólnić, możemy przekształcić go w prostą funkcję, która będzie działała na dowolnym obiekcie przekazanym jako argument i zwracała odpowiedni wynik; ponieważ pierwszy przykład wykorzystuje klasyczny wzorzec listy składanej, możemy zaoszczędzić trochę pracy, implementując go również w drugim przykładzie:

```
>>> def scramble(seq):
    res = []
    for i in range(len(seq)):
        res.append(seq[i:] + seq[:i])
    return res

>>> scramble('spam')
['spam', 'pams', 'amsp', 'mspa']

>>> def scramble(seq):
    return [seq[i:] + seq[:i] for i in range(len(seq))]

>>> scramble('spam')
['spam', 'pams', 'amsp', 'mspa']

>>> for x in scramble((1, 2, 3)):
    print(x, end=' ')
(1, 2, 3) (2, 3, 1) (3, 1, 2)
```

W zasadzie moglibyśmy tutaj użyć również rekurencji, ale w tym konkretnym kontekście prawdopodobnie byłoby to już lekką przesadą.

Funkcje generatora

Proste rozwiązywanie z poprzedniej sekcji działa, ale musi utworzyć całą listę wyników w pamięci naraz (co w przypadku bardzo dużych zestawów danych może nie być najbardziej optymalne) i wymaga od kodu wywołującego oczekiwania, aż cała lista będzie kompletna (i znów, nie jest to najlepsze rozwiązanie, jeżeli takie oczekiwanie będzie zajmowało znaczną ilość czasu). Możemy to jednak zrobić znacznie lepiej, i to na obu frontach jednocześnie, zamieniając nasz przykładowy kod na funkcję generatora, która podaje tylko jeden wynik na raz; taką funkcję możemy utworzyć na dowolny z pokazanych niżej sposobów:

```
>>> def scramble(seq):
    for i in range(len(seq)):
        seq = seq[1:] + seq[:1]
        yield seq
przypisania

>>> def scramble(seq):
    for i in range(len(seq)): # Funkcja generatora
        yield seq # Tutaj działają
```

```

        yield seq[i:] + seq[:i]                                # Zwraca jeden element
na iterację

>>> list(scramble('spam'))                               # Wywołanie list()
generuje wszystkie wyniki
['spam', 'pams', 'amsp', 'mspa']

>>> list(scramble((1, 2, 3)))                          # Działa z dowolnym
typem sekwencji
[(1, 2, 3), (2, 3, 1), (3, 1, 2)]

>>>

>>> for x in scramble((1, 2, 3)):                      # Pętla for generuje
wyniki
    print(x, end=' ')
(1, 2, 3) (2, 3, 1) (3, 1, 2)

```

Funkcje generatora w czasie działania zachowują stan lokalnego zasięgu, minimalizując wymagania dotyczące miejsca w pamięci i dzielą pracę na krótsze przedziały czasowe. W roli pełnych funkcji również mają bardzo ogólny charakter. Co ważne, pętle i inne narzędzia iteracyjne działają tak samo, niezależnie od tego, czy przechodzisz przez prawdziwą listę, czy przez generator wartości — funkcja może swobodnie wybierać między tymi dwoma schematami, a nawet zmieniać strategię działania w przyszłości.

Wyrażenia generatora

Jak już widzieliśmy, *wyrażenia generatora* — czyli złożenia w nawiasach zwykłych zamiast w kwadratowych — również generują wartości na żądanie i zachowują lokalny stan. Nie są tak elastyczne jak pełne funkcje, ale ponieważ podają swoje wartości automatycznie, w określonych przypadkach użycia wyrażenia mogą często być bardziej zwięzłe, tak jak to zostało pokazane poniżej:

```

>>> S
'spam'

>>> G = (S[i:] + S[:i] for i in range(len(S)))          # Odpowiednik w
postaci wyrażenia generatora

>>> list(G)
['spam', 'pams', 'amsp', 'mspa']

```

Zauważ, że nie możemy tutaj użyć instrukcji przypisania z pierwszej wersji funkcji generatora, ponieważ wyrażenia generatorów nie mogą zawierać instrukcji. To sprawia, że ich zakres zastosowań jest nieco węższy, choć w wielu przypadkach wyrażenia mogą wykonywać podobne zadania, tak jak to zostało pokazane poniżej. Aby uogólnić wyrażenie generatora dla dowolnego podmiotu, wystarczy z reguły opakować je w prostą funkcję, która będzie pobierała argument wywołania i zwracała generator, który go używa:

```

>>> F = lambda seq: (seq[i:] + seq[:i] for i in range(len(seq)))
>>> F(S)
<generator object <genexpr> at 0x00000000029883F0>
>>>
>>> list(F(S))

```

```

['spam', 'pams', 'amsp', 'mspa']
>>> list(F([1, 2, 3]))
[[1, 2, 3], [2, 3, 1], [3, 1, 2]]
>>> for x in F((1, 2, 3)):
    print(x, end=' ')
(1, 2, 3) (2, 3, 1) (3, 1, 2)

```

Funkcja tester

Wreszcie możemy użyć funkcji generatora lub jej odpowiednika w postaci wyrażenia generatora w funkcji tester, której używaliśmy w rozdziale 18. do wygenerowania mieszanej kolejności argumentów — funkcja mieszania kolejności elementów sekwencji staje się więc narzędziem, którego możemy używać w wielu innych kontekstach:

```

# plik scramble.py

def scramble(seq):
    for i in range(len(seq)):                      # Funkcja generatora
        yield seq[i:] + seq[:i]                    # Zwraca jeden element
    na iterację

scramble2 = lambda seq: (seq[i:] + seq[:i] for i in range(len(seq)))

```

Dzięki przeniesieniu generowania wartości do narzędzia zewnętrznego funkcja tester staje się prostsza:

```

>>> from scramble import scramble
>>> from inter2 import intersect, union
>>>
>>> def tester(func, items, trace=True):
    for args in scramble(items):                  # Użyj generatora
    (lub: scramble2(items))
        if trace: print(args)
        print(sorted(func(*args)))
>>> tester(intersect, ('aab', 'abcde', 'ababab'))
('aab', 'abcde', 'ababab')
['a', 'b']
('abcde', 'ababab', 'aab')
['a', 'b']
('ababab', 'aab', 'abcde')
['a', 'b']
>>> tester(intersect, ([1, 2], [2, 3, 4], [1, 6, 2, 7, 3]), False)
[2]
[2]

```

[2]

Permutacje: wszystkie możliwe kombinacje

Opisane techniki mają wiele innych rzeczywistych zastosowań — na przykład generowanie załączników w wiadomościach e-mail lub punktów, które mają zostać wyświetlane w interfejsie GUI. Co więcej, różne rodzaje kodowania sekwencji pełnią ważną rolę w wielu aplikacjach, od wyszukiwania po operacje matematyczne. Nasz program mieszający sekwencje to prosta zmiana kolejności elementów, ale niektóre programy oferują znacznie bardziej wyczerpujący zestaw wszystkich możliwych *permutacji* elementów, tworzonych przy użyciu funkcji rekurencyjnych z wykorzystaniem zarówno budowania list, jak i generatorów:

```
# Plik permute.py

def permute1(seq):
    if not seq:                                     # Mieszanie kolejności elementów
        dowolnej sekwencji: lista
        return [seq]                                 # Pusta sekwencja
    else:
        res = []
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]             # Usuń bieżący węzeł
            for x in permute1(rest):               # Permutuj pozostałe elementy
                res.append(seq[i:i+1] + x)          # Dodaj węzeł na początku
        return res

def permute2(seq):
    if not seq:                                     # Mieszanie dowolnej sekwencji:
        generator
        yield seq                                 # Pusta sekwencja
    else:
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]             # Usuń bieżący węzeł
            for x in permute2(rest):               # Permutuj pozostałe elementy
                yield seq[i:i+1] + x              # Dodaj węzeł na początku
```

Obie te funkcje dają takie same wyniki, chociaż druga odkłada większość pracy, dopóki nie zostanie poproszona o wynik. Przedstawiony kod jest nieco bardziej zaawansowany, szczególnie w przypadku drugiej z tych funkcji (a niektórzy mniej doświadczeni użytkownicy Pythona mogą nawet zakwalifikować go jako okrutny i nieludzki!). Jednak, jak wyjaśnimy za chwilę, zdarzają się sytuacje, w których podejście z generatorem może być bardzo przydatne.

Spróbuj samodzielnie przeanalizować kod przedstawionych przykładów. Aby ułatwić sobie prześledzenie jego działania, możesz w kluczowych miejscach wstawić polecenia `print`. Jeżeli mimo to sposób działania generatora nadal będzie dla Ciebie tajemnicą, spróbuj najpierw zrozumieć pierwszą wersję; pamiętaj, że funkcje generatora po prostu zwracają obiekty za pomocą metod, które obsługują operacje `next` wykonywane przez pętle `for` na każdym poziomie i nie dają żadnych wyników, dopóki nie zostanie wykonana iteracja; możesz również prześledzić niektóre z kolejnych przykładów, aby zobaczyć, jak korzystają z takiego kodu.

Permutacje dają więcej kombinacji niż nasz oryginalny program mieszający — dla N elementów otrzymujemy N! (silnia) wyników, a nie tylko N jak w naszym przypadku (dla czterech elementów otrzymujemy 24 kombinacje: $4! = 4 * 3 * 2 * 1$). Właśnie dlatego potrzebujemy tutaj rekurencji: liczba zagnieżdżonych pętli jest arbitralnie zmienna i zależy od długości permutowanej sekwencji:

```
>>> from scramble import scramble
>>> from permute import permute1, permute2
>>> list(scramble('abc'))                                     # Proste mieszanie: N
kombinacji
['abc', 'bca', 'cab']
>>> permute1('abc')                                         # Permutacje: N!
kombinacji
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> list(permute2('abc'))                                    # Generuj wszystkie
kombinacje
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> G = permute2('abc')                                     # Iteracje ręczne
(iter() nie jest potrzebne)
>>> next(G)
'abc'
>>> next(G)
'acb'
>>> for x in permute2('abc'): print(x)                      # Iteracje automatyczne
...wyświetla sześć wierszy...
```

Wyniki dla wersji z listą i z generatorem są takie same, ale generator minimalizuje zarówno wykorzystanie miejsca w pamięci, jak i opóźnia tworzenie wyników. W przypadku większych ilości elementów zestaw wszystkich permutacji jest znacznie większy niż w przypadku prostego mieszania:

```
>>> permute1('spam') == list(permute2('spam'))
True
>>> len(list(permute2('spam'))), len(list(scramble('spam')))
(24, 4)
>>> list(scramble('spam'))
['spam', 'pams', 'amsp', 'mspa']
>>> list(permute2('spam'))
['spam', 'spma', 'sapm', 'samp', 'smpa', 'smap', 'psam', 'psma', 'pasm',
 'pams',
 'pmsa', 'pmas', 'aspm', 'asmp', 'apsm', 'apms', 'amsp', 'amps', 'mspa',
 'msap',
 'mpsa', 'mpas', 'masp', 'maps']
```

Zgodnie z tym, co napisaliśmy w rozdziale 19., istnieją również nierekurencyjne rozwiązania alternatywne, wykorzystujące jawne stosy lub kolejki; w niektórych zastosowaniach kolejność elementów sekwencji jest z góry zdefiniowana (np. podzbiory o stałych rozmiarach i kombinacje, które odfiltrowują duplikaty o niepoprawnej kolejności). Niestety implementacja takich rozwiązań wymaga tworzenia dodatkowego, bardziej zaawansowanego kodu, dlatego nie będziemy tutaj o nich więcej pisać. Więcej informacji na ten temat możesz znaleźć w książce *Programming Python* lub po prostu eksperymentować dalej na własną rękę.

Nie nadużywaj generatorów: reguła EIBTI

Generatory są dość zaawansowanym zagadnieniem i w zasadzie mogłyby być potraktowane jako opcjonalny temat, ale zdecydowaliśmy się na ich bardziej szczegółowe omówienie ze względu na to, że są powszechnie używane w języku Python, zwłaszcza w wersjach 3.x. W rzeczywistości wielu użytkownikom wydają się zapewne mniej opcjonalne niż na przykład format Unicode (który omówimy w części VIII tej książki). Jak mogłeś się sam przekonać, wiele podstawowych, wbudowanych narzędzi Pythona, takich jak `range`, `map`, metoda `keys` słowników, a nawet pliki, jest teraz generatorami, więc musisz znać takie zagadnienia, nawet jeżeli nie piszesz własnych generatorów. Co więcej, generatory definiowane przez użytkownika są coraz bardziej powszechnie w kodzie Pythona, z którym można się dziś spotkać, na przykład w standardowej bibliotece Pythona.

Ogólnie rzecz biorąc, obowiązują tu również te same przestrogi, o których wspominaliśmy przy omawianiu list składanych: nie komplikuj kodu za pomocą generatorów zdefiniowanych przez użytkownika, jeżeli nie są one konieczne. Szczególnie w przypadku mniejszych programów i niewielkich zestawów danych korzystanie z tych narzędzi może nie być uzasadnione. W takich przypadkach wystarczą proste listy wyników; będą łatwiejsze do zrozumienia, będą automatycznie zwalniane, gdy nie będą już potrzebne, i zazwyczaj będą przetwarzane szybciej (patrz następny rozdział). Zaawansowane narzędzia, takie jak generatory oparte na ukrytej „magii”, mogą być zabawne podczas eksperymentowania, ale zazwyczaj nie powinno być dla nich miejsca w prawdziwym kodzie, z którego korzystają inni użytkownicy (chyba że ich zastosowanie jest wyraźnie uzasadnione).

Warto tutaj ponownie zacytować jedną z fundamentalnych zasad zen Pythona:

Jawne jest lepsze niż niejawne.

Powyższa zasada, często nazywana regułą EIBTI (ang. *Explicit Is Better Than Implicit*), jest jedną z podstawowych wytycznych Pythona i to nie bez powodu: im bardziej przejrzysty jest kod Twojego programu, tym większe prawdopodobieństwo, że następny programista będzie w stanie go łatwo zrozumieć. Odnosi się to bezpośrednio do generatorów, których niejawne zachowania mogą być dla wielu użytkowników znacznie trudniejsze do zrozumienia niż bardziej oczywiste rozwiązania alternatywne. Pamiętaj — nie komplikuj swojego programu, o ile nie musi być naprawdę skomplikowany!

Inne spojrzenie: miejsce i czas, zwięzłość, ekspresyjność

Biorąc pod uwagę wszystkie powyższe rozważania, musimy zauważyc, że istnieją konkretne przypadki użycia, w których zastosowanie generatorów może być naprawdę dobrym rozwiązaniem. Generatory mogą zmniejszać zużycie pamięci w niektórych programach, zmniejszać opóźnienia przetwarzania w innych, a czasami mogą wręcz umożliwić to, co bez nich było niemożliwe. Rozważmy na przykład program, który musi generować wszystkie możliwe permutacje nietrywialnej sekwencji. Ponieważ liczba kombinacji jest czynnikiem, który rośnie wykładniczo, zastosowanie omawianej wcześniej funkcji `permute1`, tworzącej listę wyników, będzie wprowadzało coraz większe opóźnienia, a może wręcz zakończyć się całkowitym niepowodzeniem ze względu na wykorzystanie wszystkich dostępnych zasobów pamięci, podczas gdy rekurencyjna funkcja `permute2` będzie nadal radziła sobie znakomicie, bardzo szybko zwracając rezultaty działania i obsługując ogromne zestawy wyników:

```

>>> import math
>>> math.factorial(10)                      # 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 *
1
3628800
>>> from permute import permute1, permute2
>>> seq = list(range(10))
>>> p1 = permute1(seq)                      # 37 sekund na komputerze z
czterordzeniowym procesorem 2GHz
                                                # Tworzy listę 3,6 miliona elementów
>>> len(p1), p1[0], p1[1]
(3628800, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 9, 8])

```

W tym przypadku funkcja tworząca listę potrzebowała na moim komputerze całych 37 sekund, aby utworzyć listę składającą się z 3,6 miliona elementów, a wersja z generatorem może zacząć zwracać wyniki natychmiast:

```

>>> p2 = permute2(seq)                      # Zwraca generator
>>> next(p2)                                # i generuje kolejne wyniki na żądanie
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> next(p2)
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]
>>> p2 = list(permute2(seq))                # Całość trwała ok. 28 sekund, co
nadal jest niepraktyczne
>>> p1 == p2                                  # Otrzymany zestaw wyników jest
identyczny jak w wersji z listą
True

```

Oczywiście możemy zoptymalizować kod tworzący listę, aby działał szybciej (np. użycie jawnego stosu zamiast rekurencji może znaczco zmienić jego wydajność), ale w przypadku większych sekwencji nie jest to żadna opcja — już przy zaledwie 50 elementach liczba permutacji praktycznie wyklucza budowanie pełnej listy wyników, a dla zwykłych śmiertelników, takich jak my, trwałyby to zdecydowanie zbyt długo (jeszcze większa liczba elementów spowodowałaby przepełnienie ustalonego limitu głębokości stosu rekurencji; patrz poprzedni rozdział). Zastosowanie generatora jest jednak nadal opłacalne — może on natychmiast wytwarzać kolejne wyniki:

```

>>> math.factorial(50)
3041409320171337804361260816606476884437764156896051200000000000000
>>> p3 = permute2(list(range(50)))
>>> next(p3)                                # permute1 tutaj nie
zadziała!
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49]

```

Aby było jeszcze zabawniej i aby uzyskać wyniki, które są bardziej zmienne i mniej oczywiste, możemy również użyć modułu `random` Pythona, opisywanego w rozdziale 5., aby losowo przetasować sekwencję, która ma być permutowana, zanim nasze funkcje `permute` zaczną działać (w rzeczywistości moglibyśmy użyć losowego tasowania elementów jako generatora permutacji — o ile możemy założyć, że w danej sesji określone tasowanie nie zostanie powtórzone, lub jeżeli będziemy sprawdzać wyniki poszczególnych tasowań i porównywać je z wcześniejszymi, aby uniknąć powtórzeń — mam nadzieję, że nie żyjemy w dziwnym wszechświecie, w którym losowa sekwencja powtarza ten sam wynik nieskończoną liczbę razy!). W przykładzie poniżej każde wywołanie funkcji `permute2` i `next` natychmiast zwraca wynik, ale funkcja `permute1` odmawia posłuszeństwa i zawiesza się:

```
>>> import random
>>> math.factorial(20)                                     # permute1 tutaj nie
zadziała!
2432902008176640000
>>> seq = list(range(20))
>>> random.shuffle(seq)                                    # Najpierw losowo
mieszamy sekwencję
>>> p = permute2(seq)
>>> next(p)
[10, 17, 4, 14, 11, 3, 16, 19, 12, 8, 6, 5, 2, 15, 18, 7, 1, 0, 13, 9]
>>> next(p)
[10, 17, 4, 14, 11, 3, 16, 19, 12, 8, 6, 5, 2, 15, 18, 7, 1, 0, 9, 13]
>>> random.shuffle(seq)
>>> p = permute2(seq)
>>> next(p)
[16, 1, 5, 14, 15, 12, 0, 2, 6, 19, 10, 17, 11, 18, 13, 7, 4, 9, 8, 3]
>>> next(p)
[16, 1, 5, 14, 15, 12, 0, 2, 6, 19, 10, 17, 11, 18, 13, 7, 4, 9, 3, 8]
```

Chodzi tutaj przede wszystkim o to, że generatorы mogą generować wyniki z dużych przestrzeni rozwiązań tam, gdzie nie mogą tego zrobić rozwiązania budujące pełne listy wyników. Z drugiej strony nie jest łatwo stwierdzić, jak często takie przypadki użycia mogą występować w zastosowaniach praktycznych, a to niekoniecznie uzasadnia użycie *niewidzialnego* sposobu generowania wartości, jaki uzyskujemy dzięki funkcjom i wyrażeniom generatora. Jak zobaczymy w części VI, generowanie wartości można również zakodować z użyciem obiektów iterowalnych opartych na *klasach*. Oparte na klasach iteratory mogą również tworzyć wyniki na żądanie i są znacznie bardziej *jawne* niż magiczne obiekty i metody tworzone dla funkcji i wyrażeń generatorów.

Nieodłączną częścią programowania jest znalezienie rozsądniego kompromisu między takimi rozwiązaniami, a tutaj nie ma żadnych bezwzględnych zasad. Chociaż zalety generatorów mogą czasem uzasadniać ich użycie, łatwość konserwacji kodu powinna zawsze być najwyższym priorytetem. Podobnie jak jest w przypadku wyrażeń, generatorы oferują także *ekspresywność* i *oszczędność* kodu, którym trudno się oprzeć, o ile naprawdę rozumiesz, jak one działają — ale zawsze powinieneś również brać pod uwagę frustrację Twoich współpracowników, którzy mogą tego nie rozumieć.

Przykład – emulowanie funkcji zip i map za pomocą narzędzi iteracyjnych

Aby pomóc Ci dalej ocenić ich rolę, przyjrzymy się jeszcze jednemu przykładowi zastosowania generatorów, który dobrze pokazuje, jak mogą one być wyraziste. Gdy poznasz listy składane, generatory i inne narzędzia iteracyjne, przekonasz się, że emulowanie wielu funkcyjnych, wbudowanych narzędzi Pythona jest zarówno proste, jak i całkiem pouczające. Na przykład mieliśmy już okazję zaobserwować, jak wbudowane funkcje `zip` i `map` łączą odpowiednio obiekty iterowalne i odwzorowania. Pobierając kilka argumentów iterowalnych, funkcja `map` wywołuje inne, określone funkcje dla poszczególnych elementów takich obiektów w sposób bardzo podobny do tego, w jaki funkcja `zip` łączy je w pary (w wersji 3.x funkcja `map` obciną krótsze iteracje; w wersji 2.x uzupełnia je obiektem `None`):

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>> list(zip(S1, S2))                                # zip łączy w pary elementy
iteratorów
[('a', 'x'), ('b', 'y'), ('c', 'z')]
# zip łączy elementy i przycina wynik do najkrótszej sekwencji
>>> list(zip([-2, -1, 0, 1, 2]))                  # Jedna sekwencja: zwraca
krotki 1-elementowe
[(-2,), (-1,), (0,), (1,), (2,)]
>>> list(zip([1, 2, 3], [2, 3, 4, 5]))            # N sekwencji: zwraca krotki
N-elementowe
[(1, 2), (2, 3), (3, 4)]
# map przekazuje połączone elementy do funkcji, przycina do najkrótszej
sekwiencji
>>> list(map(abs, [-2, -1, 0, 1, 2]))            # Jedna sekwencja: wywołanie
funkcji 1-argumentowej
[2, 1, 0, 1, 2]
>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5]))      # N sekwencji: wywołanie
funkcji N-argumentowej (3.x)
[1, 8, 81]
# map i zip akceptują dowolne obiekty iterowalne
>>> list(map(lambda x, y: x + y, open('script2.py'), open('script2.py')))
['import sys\nimport sys\n', 'print(sys.path)\nprint(sys.path)\n', ...itd...]
>>> [x + y for (x, y) in zip(open('script2.py'), open('script2.py'))]
['import sys\nimport sys\n', 'print(sys.path)\nprint(sys.path)\n', ...itd...]
```

Chociaż mechanizmy te są wykorzystywane do różnych celów, to jeżeli przyjrzyisz się uważnie tym przykładom, z pewnością zauważysz związek między wynikami działania funkcji `zip` i argumentami funkcji odwzorowanych, który będziemy wykorzystywać w naszym następnym przykładzie.

Tworzymy własną implementację funkcji map

Choć funkcje wbudowane `map` i `zip` są szybkie i wygodne w użyciu, zawsze istnieje możliwość emulowania ich na własną rękę. W poprzednim rozdziale na przykład mieliśmy okazję stworzyć funkcję emulującą funkcję wbudowaną `map`, obsługującą tylko jeden argument sekwencji (lub innego obiektu iterowalnego). Uzupełnienie tej implementacji o obsługę większej liczby sekwencji nie stanowi jednak problemu.

```
# map(func, seqs...): emulacja z użyciem funkcji zip
def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res
print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

Ta wersja intensywnie wykorzystuje specjalną składnię `*args` służącą do przekazywania argumentów między wywoływanymi funkcjami. Funkcja rozpakowuje swoje argumenty (będące sekwencjami, a właściwie obiektami iterowanymi) za pomocą funkcji `zip`, łącząc je w nowe sekwencje, po czym wywołuje funkcję przekazaną w pierwszym argumencie z użyciem tych nowych sekwencji jako argumentów. Wykorzystujemy zaobserwowany wyżej fakt, że operacja `zip` jest w zasadzie zagnieżdżoną operacją `map`. Test poniżej definicji funkcji sprawdza działanie naszego generatora dla dwóch sekwencji (wyniki są identyczne z tymi, które zwróciłaby wbudowana funkcja `map`). Kod tego przykładu znajdziesz w pliku `mymap.py` w pakietie kodów do książki:

```
[2, 1, 0, 1, 2]
[1, 8, 81]
```

W rzeczywistości ta wersja wykorzystuje klasyczny wzorzec znany z *list składanych*, budując listę wyników w pętli `for`. Naszą funkcję `map` możemy zatem zbudować w sposób bardziej zwarty, wykorzystując jednowierszowe wyrażenie listy składanej.

```
# Użycie listy składanej
def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]
print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

Uruchomione testy zwracają taki sam wynik jak poprzednio, ale kod jest krótszy i prawdopodobnie będzie szybszy (więcej informacji na temat pomiarów wydajności przedstawię w punkcie „Pomiary wydajności implementacji iteratorów” w dalszej części rozdziału). Obie z przedstawionych wersji funkcji `mymap` składają wynik w całości, co może mieć konsekwencje w postaci większego zapotrzebowania na pamięć. Dzięki możliwościom *funkcji i wyrażeń generatorów* możemy w prosty sposób przepisać te funkcje tak, aby zwracały wyniki na żądanie.

```
# Użycie generatorów: yield i ...
def mymap(func, *seqs):
    for args in zip(*seqs):
```

```

        yield func(*args)

def mymap(func, *seqs):
    return (func(*args) for args in zip(*seqs))

```

Te wersje nie tworzą całych wyników naraz, ale zwracają generatory obsługujące protokół iteracyjny: pierwsza wersja zwraca wyniki za pomocą instrukcji `yield`, a druga jest jej funkcjonalnym odpowiednikiem dzięki wyrażeniu generatora. Obie wersje zwrócią takie same wyniki, jeżeli przekształcimy je na listy za pomocą funkcji `list`.

```

print(list(mymap(abs, [-2, -1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))

```

Jednak w tym przypadku różnica jest taka, że generatory nie wykonują żadnej pracy do momentu, gdy zostaje na nich wywołana funkcja `list` aktywująca protokół iteracyjny. Generatory zwrócone z tych funkcji, jak również wersja dla Pythona 3.x wykorzystująca funkcję `zip`, generują wyniki na żądanie.

Własna wersja funkcji `zip(...)` i `map(None, ...)`

Oczywiście spora część magii ukrytej w powyższych przykładach działa dzięki zastosowaniu wbudowanej funkcji `zip` łączącej elementy z kilku sekwencji lub z innych obiektów iterowalnych. Warto również zauważyć, że nasze implementacje funkcji `map` w rzeczywistości kopią zachowanie funkcji `map` z Pythona 3.x: przycinają wynik do długości najkrótszego argumentu i nie obsługują dopełniania wyników obiektami `None`, tak jak to robi funkcja `map` w Pythonie 2.x:

```

C:\code> c:\python27\python
>>> map(None, [1, 2, 3], [2, 3, 4, 5])
[(1, 2), (2, 3), (3, 4), (None, 5)]
>>> map(None, 'abc', 'xyz123')
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]

```

Wykorzystując narzędzia iteracyjne, możemy zaimplementować własną wersję funkcji `zip` przycinającej wynik do najkrótszej sekwencji lub funkcji `map` uzupełniającej brakujące elementy sekwencji obiektami `None` (w wersji 2.x). Jak się okazuje, funkcje te będą bardzo do siebie podobne:

```

# Własne implementacje funkcji zip(seqs...) oraz map(None, seqs...) w wersji 2.x

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res = []

```

```

while any(seqs):
    res.append(tuple((S.pop(0) if S else pad) for S in seqs))
return res

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))

```

Obie przedstawione funkcje współpracują z dowolnym obiektem *iterowalnym*, ponieważ przekształcają swoje argumenty na listę, wymuszając wykonanie generatora (czyli jako argumenty mogą być podane pliki lub inne sekwencje, jak ciągi znaków). Należy zwrócić uwagę na użycie funkcji `all` i `any`: zwracają wartość `True`, jeżeli odpowiednio wszystkie lub dowolny element obiektu iterowanego ma wartość `True` (lub jest niepusty). Te wbudowane funkcje służą do zatrzymania działania pętli, jeżeli dowolny lub wszystkie elementy sekwencji zostaną usunięte.

Warto również zwrócić uwagę na argument *mogący być tylko słowem kluczowym* `pad`, dostępny wyłącznie w wersji 3.x. W przeciwieństwie do standardowej funkcji `zip` z wersji 2.x nasza funkcja akceptuje dowolne obiekty uzupełniające sekwencje o mniejszej długości (jeżeli używasz Pythona 2., powinieneś użyć argumentów typu `**kargs`, szczegóły tej techniki znajdziesz w rozdziale 18.). Po wywołaniu tych funkcji zostają wyświetlane następujące wyniki (`zip` i dwa wyniki `map` z uzupełnianiem):

```

[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]

```

Omawiane funkcje nie mogą być w prosty sposób zaimplementowane w formie list składanych, ponieważ ich pętle `for` są zbyt specyficzne. Jednak, podobnie jak poprzednio, nasze wersje funkcji `zip` i `map`, które obecnie zwracają listę wyników w całości, mogą być w prosty sposób przekształcone w *generatory* (przy wykorzystaniu instrukcji `yield`), co spowoduje, że każdy element wyniku będzie generowany na żądanie. Efekt będzie zbliżony do poprzedniego, z tą różnicą, że w celu wyświetlenia wyników musimy przekształcić generator na listę:

```

# Użycie generatorów: yield

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    while any(seqs):
        yield tuple((S.pop(0) if S else pad) for S in seqs)

S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2)))
print(list(mymapPad(S1, S2)))

```

```
print(list(mymapPad(S1, S2, pad=99)))
```

Na koniec przedstawiamy alternatywną implementację naszych emulatorów funkcji `zip` i `map`: zamiast usuwać elementy z list za pomocą metody `pop`, funkcje wykonują swoje zadanie, obliczając minimalną i maksymalną *długość argumentów*. Dzięki tym wartościom łatwo jest utworzyć odpowiednio zagnieżdżone listy składane, przechodzące przez kolejne zakresy indeksów argumentów.

```
# Alternatywna implementacja wykorzystująca długości

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return [tuple(S[i] for S in seqs) for i in range(minlen)]

def mymapPad(*seqs, pad=None):
    maxlen = max(len(S) for S in seqs)
    index = range(maxlen)
    return [tuple((S[i] if len(S) > i else pad) for S in seqs) for i in index]

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

Z powodu wykorzystania długości i dostępu do elementów po indeksach funkcje te zakładają, że mają do czynienia z *sekwencjami* lub z obiektami o zbliżonych cechach, ale nie mogą to być zupełnie dowolne obiekty iterowalne. Zewnętrzna lista składana iteruje po zakresie indeksów argumentów, wewnętrzna (przekształcona na krotkę) ponownie iteruje po przekazanych argumentach, wydobywając elementy na odpowiadających sobie indeksach. Wyniki tych funkcji są takie same jak poprzedniej implementacji.

W tym przykładzie najbardziej rzuca się w oczy powszechnie wykorzystanie generatorów i iteratorów. Argumenty przekazane do funkcji `min` i `max` są wyrażeniami generatorów, które są wykonywane do końca, zanim zostaną uruchomione zagnieżdżone listy składane. Co więcej, zagnieżdżone listy składane wykorzystują dwa poziomy opóźnionego wyliczenia: w Pythonie 3.x funkcja wbudowana `range` jest obiektem iterowalnym, jest nim również wyrażenie generatora przekształcone na krotkę.

W rzeczywistości w tych funkcjach nie dochodzi do jakiegokolwiek wyliczenia wartości aż do momentu, gdy nawiasy kwadratowe wyrażeń list składanych wymuszą wygenerowanie listy: efektywnie powodują pełne wywołanie sekwencji generatorów. Aby również to wyrażenie przekształcić na generator, zamiast budować listę w całości, wystarczy nawiasy kwadratowe zastąpić okrągłymi. Oto odpowiedni fragment naszej wersji funkcji `zip`:

```
# Użycie generatorów: (...)

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return (tuple(S[i] for S in seqs) for i in range(minlen))

S1, S2 = 'abc', 'xyz123'
```

```
print(list(myzip(S1, S2))) # Jedziemy!... [('a',  
'x'), ('b', 'y'), ('c', 'z')]
```

W tym przypadku, aby aktywować generatorы i inne obiekty iterowalne i spowodować wyświetlenie wyników na ekranie, musimy posłużyć się wywołaniem funkcji `list`. Zachęcam do własnych eksperymentów z tymi funkcjami w celu pogłębienia wiedzy. Opracowanie bardziej rozbudowanych odpowiedników obu funkcji pozostawimy jako sugerowane ćwiczenie do samodzielnego wykonania (zobacz ramkę „Warto pamiętać: iteratory jednoprzebiegowe”).

Warto pamiętać: iteracje jednoprzebiegowe

W rozdziale 14. poznaliśmy funkcje wbudowane (takie jak `map`) obsługujące tylko jeden przebieg iteracji, po którym ich wyniki są puste. Obiecałem, że w dalszej części książki zamieścimy przykład wyjaśniający, dlaczego to zjawisko jest bardzo subtelne, ale jednocześnie istotne w praktyce. W tym miejscu książki mamy już za sobą solidne podstawy zagadnień związanych z iteracjami, nadszedł więc dobry moment, by spełnić obietnicę. Weźmy na przykład następującą alternatywę dla przykładów emulacji funkcji `zip` przedstawionych w tym rozdziale (przykład zaczerpnięty z jednego z podręczników Pythona):

```
def myzip(*args):  
    iters = map(iter, args)  
    while iters:  
        res = [next(i) for i in iters]  
        yield tuple(res)
```

Dzięki temu, że ten kod wykorzystuje funkcje `iter` oraz `next`, może współpracować z dowolnym obiektem iterowalnym. Warto zwrócić uwagę, że nie ma sensu przechwytywanie wyjątku `StopIteration` wywoływanego przez funkcję `next(i)`: jej wywołanie zostanie przekazane przez funkcję, co w efekcie spowoduje zakończenie pracy generatora — tak samo, jak gdybyśmy zwróciли wartość za pomocą instrukcji `return`. Instrukcja `while iters:` wystarcza do realizacji pętli po elementach argumentów, pod warunkiem że do funkcji został przekazany co najmniej jeden. Zapobiega też efektowi nieskończonej pętli, jeżeli nie został podany żaden argument (wyrażenie listy składanej zwróci pustą listę).

Powyższy kod bez problemu działa w Pythonie 2.x:

```
>>> list(myzip('abc', 'lmnop'))  
[('a', 'l'), ('b', 'm'), ('c', 'n')]
```

Jednak w Pythonie 3.x wpadamy w pętlę nieskończoną, ponieważ funkcja `map` w wersji 3.x zwraca generator jednorazowy zamiast listy, jak to ma miejsce w 2.x. W wersji 3.x po pierwszym przejściu iteratatora przez obiekt `iters` zostanie wyczerpany, ale nadal będzie miał wartość `True` (a `res` będzie miał wartość `[]`). Aby nasza funkcja zadziałała w wersji 3.x, musimy posłużyć się funkcją wbudowaną `list` do utworzenia obiektu obsługującego wielokrotną iterację:

```
def myzip(*args):  
    iters = list(map(iter, args)) # Zezwala na iteracje  
    wieloprzebiegowe  
    ...dalszy ciąg taki sam...
```

Warto prześledzić ten kod, aby przekonać się, w jaki sposób działa. Morał z tej opowieści: w 3.x przekształcenie rezultatów działania funkcji `map` na listę służy nie tylko do wyświetlania wyników na ekranie!



Więcej przykładów zastosowania polecenia `yield` znajdziesz w rozdziale 30., gdzie użyjemy go w połączeniu z metodą `__iter__` do zaimplementowania zdefiniowanych przez użytkownika obiektów iterowalnych w zautomatyzowany sposób. Zachowanie stanu zmiennych lokalnych w tej roli służy jako alternatywa dla atrybutów klas, podobnie jak robią to funkcje domknięcia, które omawialiśmy w rozdziale 17.; jak się przekonasz, ta technika łączy klasy i narzędzia funkcyjne zamiast stanowić alternatywę dla całego paradygmatu.

Podsumowanie obiektów składanych

W tym rozdziale skupialiśmy się na listach składanych i generatorach, jednak należy pamiętać, że istnieją jeszcze dwie inne formy wyrażeń składanych, dostępne w wersji 3.x i 2.x: zbiorы и словари składane. Mieliśmy okazję spotkać się z nimi w rozdziałach 5. i 8., ale teraz, wzbogaceni o większą wiedzę na temat obiektów składanych i generatorów, możemy w pełni docenić możliwości tych rozszerzeń Pythona.

- W przypadku *zbiorów* nowy literał `{1, 2, 3}` jest równoważny wyrażeniu `set([1, 2, 3])`, a nowa składnia zbiorów składanych `{f(x) for x in S if P(x)}` działa tak samo jak wyrażenie generatora `set(f(x) for x in S if P(x))`, gdzie `f(x)` jest dowolnym wyrażeniem.
- W przypadku *słowników* nowa składnia słowników składanych `{key: val for (key, val) in zip(keys, vals)}` działa tak samo jak wyrażenie `dict(zip(keys, vals))`, a `{x: f(x) for x in items}` działa jak wyrażenie generatora `dict((x, f(x)) for x in items)`.

Poniższy listing prezentuje podsumowanie wszystkich wyrażeń obiektów składanych w Pythonie 3.x i 2.7. Ostatnie dwie konstrukcje są nowe i nie są dostępne w wersji 2.6 i wcześniejszych.

```
>>> [x * x for x in range(10)]          # Lista składana: zwraca listę
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81] # Odpowiednik listy (wyrażenie
                                         generatora)

>>> (x * x for x in range(10))        # Wyrażenie generatora: zwraca elementy
                                         na żądanie

<generator object at 0x009E7328>      # W niektórych kontekstach nawiasy są
                                         opcjonalne

>>> {x * x for x in range(10)}        # Zbiór składany, dostępny w wersjach
                                         3.x i 2.7

{0, 1, 4, 81, 64, 9, 16, 49, 25, 36} # {x, y} to literał zbioru w tych
                                         wersjach

>>> {x: x * x for x in range(10)}     # Słownik składany, dostępny w wersjach
                                         3.x i 2.7

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Zakresy i zmienne składane

Teraz gdy widzieliśmy już wszystkie formy wyrażeń składanych, powinieneś powrócić na chwilę do rozdziału 17. i przejrzeć zagadnienia związane z lokalizacją zmiennych pętli w takich wyrażeniach. Python 3.x lokalizuje zmienne pętli we wszystkich czterech formach wyrażeń —

tymczasowe nazwy zmiennych pętli w generatorach, zbiorach, słownikach i listach są lokalne dla danego wyrażenia. Nie kolidują w ten sposób z nazwami na zewnątrz, ale nie są tam również dostępne i działają inaczej niż instrukcja pętli `for`:

```
c:\code> py -3
>>> (X for X in range(5))
<generator object <genexpr> at 0x00000000028E4798>
>>> X
NameError: name 'X' is not defined
>>> X = 99
>>> [X for X in range(5)]          # 3.x: generatory, zbiory,
słowniki i listy lokalizują zmienne
[0, 1, 2, 3, 4]
>>> X
99
>>> Y = 99
>>> for Y in range(5): pass      # Ale już w pętlach zmienne nie
są lokalizowane
>>> Y
4
```

Jak wspominaliśmy w rozdziale 17., w wersji 3.x zmienne przypisane w wyrażeniach składanych są tak naprawdę specjalnym, zagnieżdżonym zasięgiem; inne nazwy przywoływanie w tych wyrażeniach są zgodne ze zwykłymi regułami LEGB. Na przykład w poniższym generatorze zmienna `Z` jest zlokalizowana, ale `Y` i `X` znajdują się w otaczających zasięgach lokalnych i globalnych, jak zwykle:

```
>>> X = 'aaa'
>>> def func():
    Y = 'bbb'
    print('.join(Z for Z in X + Y))'      # Z – wyrażenie złożone, Y –
zmienna lokalna, X – zmienna globalna
>>> func()
aaabbb
```

Pod tym względem Python 2.x zachowuje się niemal tak samo, z tą tylko różnicą, że zmienne *list składanych* nie są lokalizowane — działają one tak samo jak w pętli `for` i zachowują swoje ostatnie wartości iteracji, ale dzięki temu są również potencjalnie narażone na nieoczekiwane konflikty z nazwami zewnętrznymi. Wyrażenia generatora, zbiorów i słowników lokalizują nazwy jak w wersji 3.x:

```
c:\code> py -2
>>> (X for X in range(5))
<generator object <genexpr> at 0x0000000002147EE8>
>>> X
```

```

NameError: name 'X' is not defined
>>> X = 99
>>> [X for X in range(5)]           # 2.x: Listy nie lokalizują
swoich zmiennych, tak jak pętle for
[0, 1, 2, 3, 4]
>>> X
4
>>> Y = 99
>>> for Y in range(5): pass        # Pętle for nie lokalizują
zmiennych ani w wersji 2.x, ani w 3.x
>>> Y
4

```

Jeżeli zależy Ci na przenośności wersji, powinieneś po prostu używać w wyrażeniach składanych unikalnych nazw zmiennych. Zachowanie wersji 2.x ma sens, biorąc pod uwagę fakt, że obiekt generatora jest odrzucany po zakończeniu tworzenia wyników, ale listy składane są odpowiednikiem pętli `for` — choć powyższe stwierdzenia nie dotyczą zbiorów ani słowników składanych, które lokalizują nazwy w obu liniach Pythona i są, poniekąd całkiem przypadkowo, tematem następnej sekcji.

Zrozumieć zbiory i słowniki składane

W pewnym sensie zbiory i słowniki składane stanowią jedynie składniowe uproszczenie istniejących konstrukcji, które można zbudować, na przykład przekształcając wyrażenie generatora na odpowiedni typ. Zarówno zbiory, jak i słowniki składane mogą pracować z dowolnym obiektem iterowalnym, więc również generator sprawdzi się w tej roli:

```

>>> {x * x for x in range(10)}          # Zbiór składany
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> set(x * x for x in range(10))       # Generator i nazwa typu
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> {x: x * x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> dict((x, x * x) for x in range(10))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> x                                     # Zmienna pętli jest
lokalizowana w wersjach 2.x i 3.x
NameError: name 'x' is not defined

```

Podobnie jak w przypadku listy składanej, wynik możemy wygenerować za pomocą odpowiedniej pętli. Oto rozwinięcia powyższych obiektów składanych w postaci zwykłej instrukcji pętli (choć różnią się sposobem lokalizacji nazw zgodnie z tym, co pisaliśmy w poprzedniej sekcji):

```
>>> res = set()
```

```

>>> for x in range(10):                      # Odpowiednik zbioru składanego
    res.add(x * x)

>>> res
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> res = {}

>>> for x in range(10):                      # Odpowiednik słownika składanego
    res[x] = x * x

>>> res
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>> x                                         # Nazwa lokalizowana w wyrażeniach
złożonych, ale nie w pętlach

9

```

Warto zwrócić uwagę, że choć zarówno zbiór, jak i słownik akceptują obiekty iterowalne, nie mają pojęcia o *generowaniu* wyników na żądanie — obie formy tworzą od razu kompletne obiekty wynikowe. Jeżeli chciałbyś tworzyć klucze i wartości na żądanie, bardziej odpowiednim rozwiązańiem będzie wyrażenie generatora:

```

>>> G = ((x, x * x) for x in range(10))
>>> next(G)
(0, 0)
>>> next(G)
(1, 1)

```

Rozszerzona składnia zbiorów i słowników składanych

Zbiory i słowniki składane, podobnie jak listy składane i wyrażenia generatorów, obsługują zagnieżdżone klauzule `if` pozwalające odfiltrować z wyników wybrane elementy — poniżej zamieszczamy przykład kodu zwracającego wartości kwadratów parzystych elementów zakresu (czyli elementów, które nie posiadają reszty z dzielenia przez dwa):

```

>>> [x * x for x in range(10) if x % 2 == 0]          # Listy zachowują
kolejność elementów
[0, 4, 16, 36, 64]

>>> {x * x for x in range(10) if x % 2 == 0}        # Zbiory nie
{0, 16, 4, 64, 36}

>>> {x: x * x for x in range(10) if x % 2 == 0}      # Ani klucze słowników
{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}

```

Zagnieżdżone pętle `for` również działają bez problemów, choć nieuporządkowana i unikatowa pod względem elementów natura obu typów obiektów może powodować, że wyniki będą nieco mniej czytelne:

```
>>> [x + y for x in [1, 2, 3] for y in [4, 5, 6]]      # Listy zachowują
       duplikaty
[5, 6, 7, 6, 7, 8, 7, 8, 9]
>>> {x + y for x in [1, 2, 3] for y in [4, 5, 6]}      # Zbiory nie
{8, 9, 5, 6, 7}
>>> {x: y for x in [1, 2, 3] for y in [4, 5, 6]}      # Ani klucze słowników
{1: 6, 2: 6, 3: 6}
```

Zbiory i słowniki składane, podobnie jak listy składane i wyrażenia generatorów, mogą wykorzystywać dowolne obiekty iterowalne: listy, ciągi znaków, pliki, zakresy i dosłownie wszystkie inne typy obiektów, które obsługują protokół iteracyjny.

```
>>> {x + y for x in 'ab' for y in 'cd'}
{'ac', 'bd', 'bc', 'ad'}
>>> {x + y: (ord(x), ord(y)) for x in 'ab' for y in 'cd'}
{'ac': (97, 99), 'bd': (98, 100), 'bc': (98, 99), 'ad': (97, 100)}
>>> {k * 2 for k in ['mielonka', 'szynka', 'kiełbasa'] if k[1] == 'i'}
{'mielonkamielonka', 'kiełbasakiełbasa'}
>>> {k.upper(): k * 2 for k in ['mielonka', 'szynka', 'kiełbasa'] if k[1] ==
'i'}
{'KIEŁBASA': 'kiełbasakiełbasa', 'MIELONKA': 'mielonkamielonka'}
```

Czytelników zainteresowanych dalszymi szczegółami zachęcam do samodzielnego eksperymentowania. Zbiory i słowniki składane mogą w niektórych przypadkach działać wydajniej w porównaniu z analogicznymi generatorami lub pętlami `for`, ale nie jest to zasadą. Aby zyskać pewność, należy po prostu zmierzyć wydajność każdego z tych rozwiązań — co w dosyć naturalny sposób prowadzi nas do tematyki następnego rozdziału.

Podsumowanie rozdziału

W tym rozdziale podsumowaliśmy omawiane dotychczas zagadnienia dotyczące wbudowanych narzędzi wykorzystujących wyrażenia składane oraz iteracje. Omówiliśmy listy składane w kontekście narzędzi funkcyjnych, a także zaprezentowaliśmy funkcje i wyrażenia generatora jako dodatkowe narzędzia protokołu iteracyjnego. Na zakończenie podsumowaliśmy także cztery formy wyrażeń składanych, dostępnych obecnie w Pythonie — listy, generatory, zbiory i słowniki. Chociaż poznaliśmy już wszystkie wbudowane narzędzia iteracyjne, jeszcze raz powrócimy do tego tematu w rozdziale 30. podczas omawiania definiowanych przez użytkownika obiektów iterowalnych opartych na klasach.

Następny rozdział jest kontynuacją tych zagadnień — uzupełnia tę część książki poprzez studium przypadku, w którym będziemy w praktyczny sposób zajmować się wydajnością poznanych wcześniej narzędzi. Zanim jednak przejdziemy do testowania wydajności wyrażeń złożonych i generatorów, rozwiązanie quizu z tego rozdziału da Ci niepowtarzalną szansę na sprawdzenie, czego się tutaj dowiedziałeś.

Sprawdź swoją wiedzę — quiz

1. Jaka jest różnica między nawiasami kwadratowymi a okrągłymi w wyrażeniu listy składanej?
2. Jaki jest związek między generatorami a iteratorami?
3. W jaki sposób rozpoznać, czy funkcja jest generatorem?
4. Co robi instrukcja `yield`?
5. Jaki jest związek między funkcją `map` a listami składanymi? Podaj podobieństwa i różnice.

Sprawdź swoją wiedzę — odpowiedzi

1. Wyrażenie listy składanej w nawiasach kwadratowych zwraca listę wygenerowaną w całości. To samo wyrażenie ujęte w nawiasy okrągłe jest w rzeczywistości wyrażeniem generatora — efekt działania jest zbliżony, ale wynik nie jest tworzony od razu w całości. Zamiast tego wyrażenie generatora zwraca obiekt generatora, który zwraca kolejne wyniki na żądanie, kiedy zostanie użyty w kontekście iteracyjnym.
2. Generatory są obiektami automatycznie obsługującymi protokół iteracyjny: posiadają iteratory z metodą `__next__` (`next` w wersji 2.x) zwracającą kolejny element z serii wyników i zgłaszającą wyjątek sygnalizujący osiągnięcie końca serii. W Pythonie możemy kodować funkcje generatora za pomocą polecień `def` i `yield`, wyrażenia generatora za pomocą wyrażeń składanych umieszczonych w nawiasach oraz obiekty generatora za pomocą klas, które definiują specjalną metodę o nazwie `__iter__` (omówioną w dalszej części książki).
3. Funkcja generatora wykorzystuje w swoim kodzie instrukcję `yield`. Funkcje generatorów są pod względem składniowym identyczne ze zwykłymi funkcjami, ale są komplikowane przez interpreter Pythona w taki sposób, aby po wywołaniu zwracały obiekt iterowalny, który zachowuje stan i lokalizację w kodzie pomiędzy kolejnymi wywołaniami.
4. Gdy instrukcja `yield` występuje w kodzie funkcji, interpreter Pythona kompiluje tę funkcję w specjalny sposób, aby przy wywołaniu zwracała generator. Obiekt generatora zwracany przez funkcję tego typu obsługuje protokół iteracyjny. Gdy zostaje wywołana instrukcja `yield`, funkcja zwraca wynik do wyrażenia wywołującego i zawiesza swoje działanie. Funkcja może zostać wznowiona od tego miejsca, w którym wywołała instrukcję `yield` — służy do tego funkcja wbudowana `next` lub metoda `__next__` generatora. W bardziej zaawansowanych zastosowaniach metoda `send` w podobny sposób wznowia działanie generatora, ale może także przekazać wartość, która pojawi się jako wartość wyrażenia `yield`. Funkcje generatora mogą mieć także instrukcję `return`, która kończy działanie generatora.
5. Funkcja `map` ma działanie podobne do listy składanej: obie konstrukcje budują nową listę, zbierając wyniki operacji wykonanych kolejno na elementach na sekwencji lub innego obiektu iterowalnego. Główna różnica polega na tym, że funkcja `map` wymaga wywołania wskazanej funkcji na każdym elemencie sekwencji, natomiast lista składana pozwala wywoływać dowolne wyrażenie. Dzięki temu listy składane mają bardziej ogólne zastosowanie; można ich użyć do wywołania funkcji na każdym

elemencie (jak ma to miejsce w przypadku `map`), ale do wywołania innych wyrażeń za pomocą `map` użytkownik musi napisać do tego funkcję. Listy składane obsługują ponadto składnię rozszerzoną, pozwalającą na zastosowanie zagnieżdżonych pętli `for` i klauzuli `if` pozwalającej na odfiltrowanie elementów z wyników (co jest odpowiednikiem funkcji wbudowanej `filter`). W Pythonie 3.x funkcja `map` wyróżnia się również tym, że tworzy *generator* wartości, podczas gdy listy składane generują od razu pełną listę wszystkich wyników. W wersji 2.x oba narzędzia tworzą pełne listy wyników.

[1] Technicznie rzecz biorąc, w wersjach 2.x i we wszystkich wersjach linii 3.x aż do 3.3 Python traktuje wartości zwarcane w funkcjach generatora przez instrukcję `return` jako błędy składniowe. Począwszy od wersji 3.3, wartość zwarcana przez instrukcję `return` jest dozwolona i dołączana do obiektu `StopIteration`, z tym że wartość ta jest ignorowana w kontekście automatycznych iteracji, a użycie tej instrukcji powoduje, że taki kod jest niezgodny ze wszystkimi wcześniejszymi wersjami Pythona.

[2] Co ciekawe, funkcje generatorów można wykorzystać również jako bardzo uproszczony mechanizm *współbieżności*: pozwalają na wykonanie kilku operacji naprzemiennie, z możliwością przełączania kontekstu między nimi w miejscu wywołania instrukcji `yield`. Generatory nie są jednak wątkami, program nadal jest wykonywany w jednym wątku fizycznym. Obsługa wątków jest mechanizmem bardziej ogólnym (procedury mogą działać naprawdę równolegle i zapisywać swoje wyniki w kolejce), ale generatorów tworzy się znacznie prościej. Krótkie wprowadzenie do mechanizmów programowania wielowątkowego w Pythonie można znaleźć w drugim przypisie w rozdziale 17. Należy też pamiętać, że przełączanie kontekstu odbywa się w *stałych punktach*, to znaczy w miejscu wywołania instrukcji `yield` oraz w miejscu ponownego wywołania generatora (`next`), przez co generatorów nie są mechanizmami w pełni współbieżnymi. Ich działanie jest natomiast zbliżone do *współprogramów* (ang. *coroutines*). Jest to formalny koncept programowania równoległego, którego tematyka wykracza jednak daleko poza zakres tego rozdziału.

Rozdział 21. Wprowadzenie do pomiarów wydajności

Teraz gdy wiemy już, jak tworzyć funkcje i narzędzia iteracyjne, zejdziemy nieco w bok od naszej głównej ścieżki i spróbujemy się przekonać, jak oba rodzaje tych narzędzi działają w praktyce. Niniejszy rozdział zamyka część funkcyjną tej książki większym studium przypadku, w którym będziemy mierzyć względną wydajność narzędzi iteracyjnych poznanych do tej pory.

We wspomnianym studium przypadku będziemy badać narzędzia do pomiaru czasu w Pythonie, ogólnie omawiać techniki mierzenia wydajności i odkrywać kod, który jest nieco bardziej realistyczny i użyteczny niż większość tego, co widzieliśmy do tej pory. Będziemy również mierzyć szybkość bieżących implementacji Pythona — jest to element, który może, ale nie musi być istotny, w zależności od tego, nad jakimi rodzajami kodu aktualnie pracujesz.

Wreszcie, ponieważ jest to ostatni rozdział w tej części książki, zakończymy zwyklimi zestawami „pułapek” i ćwiczeń, które pomogą Ci zacząć wdrażać w praktyce pomysły, o których tutaj czytałeś. Najpierw jednak zabawimy się trochę z konkretną aplikacją napisaną w Pythonie.

Pomiary wydajności iteracji

W tej książce poznaliśmy sporo alternatywnych sposobów implementowania iteracji. Podobnie jak w przypadku programowania, reprezentują one różnego rodzaju kompromisy — zarówno pod względem czynników subiektywnych, takich jak ekspresywność, jak i bardziej obiektywnych kryteriów, takich jak wydajność. Część zadań programisty oraz inżyniera oprogramowania polega właśnie na wybieraniu odpowiednich narzędzi na podstawie takich czynników.

Jeżeli chodzi o wydajność, kilka razy już wspomniałem o tym, że listy składane mają czasami znaczną przewagę szybkości nad pętlami `for` oraz że wywołania funkcji `map` mogą być szybsze lub wolniejsze niż te oba rozwiązań, w zależności od użytego wzorca wywołania. Funkcje i wyrażenia generatora opisywane w poprzednim rozdziale są zwykle nieco wolniejsze w działaniu od list składanych, ale za to pozwalają zminimalizować zużycie pamięci i nie wymuszają na wywołującym kodzie konieczności oczekiwania na wygenerowanie wszystkich rezultatów, co może mieć ogromne znaczenie przy dużej liczbie wyników.

Są to oczywiście pewne uogólnienia, które nadal są prawdziwe, ale względna wydajność poszczególnych rozwiązań może się różnić w zależności od wersji Pythona, ponieważ język ten jest nieustannie modyfikowany i optymalizowany, a struktura kodu może w niemal dowolny sposób wpływać na jego szybkość działania. Jeżeli chcesz sam przekonać się, jaką mają wydajność, powinieneś samodzielnie sprawdzić konkretne rozwiązania na swoim komputerze i posiadanej wersji Pythona.

Moduł pomiaru czasu domowej roboty

Na szczęście Python ułatwia tworzenie kodu mającego na celu pomiary czasu. Na przykład, aby uzyskać całkowity czas potrzebny na wykonanie wielu wywołań funkcji z arbitralnymi argumentami pozycyjnymi, wystarczająca może być następująca, prosta funkcja:

```
# Plik timer0.py
import time
def timer(func, *args):                      # Uproszczona funkcja pomiaru czasu
    start = time.clock()
    for i in range(1000):
        func(*args)
    return time.clock() - start               # Całkowity czas, który upłynął [w sekundach]
```

Takie rozwiązanie działa — funkcja odczytuje czas uruchomienia z modułu `time` Pythona, następnie 1000 razy wywołuje przekazaną funkcję z przekazanymi do niej argumentami, a po zakończeniu odejmuje czas rozpoczęcia od czasu zakończenia działania. Na moim komputerze w Pythonie 3.3 wyglądało to tak:

```
>>> from timer0 import timer
>>> timer(pow, 2, 1000)                      # Czas wywołania 1000 razy funkcji pow(2, 1000)
0.00296260674205626
```

```
>>> timer(str.upper, 'spam')                                # Czas wywołania 1000 razy funkcji 'spam'.upper()
0.0005165746166859719
```

Takie rozwiązanie pomiaru czasu, choć proste, jest również dość ograniczone i celowo wykazuje pewne klasyczne błędy zarówno w projektowaniu funkcji, jak i testowaniu. Należą do nich między innymi:

- Brak obsługi argumentów *słów kluczowych* w testowanym wywołaniu funkcji.
- Zakodowana na sztywno *liczba wywołań* testowanej funkcji.
- Czas wykonania funkcji `range` jest doliczany do ogólnego czasu działania testowanej funkcji.
- Zawsze używamy metody `time.clock`, co może nie być najlepszym rozwiązaniem poza systemem Windows.
- Nie pozwala kodowi wywołującemu na sprawdzenie, czy testowana funkcja rzeczywiście działała.
- Podaje tylko *całkowity* czas działania, który może się zmieniać w przypadku mocno obciążonych komputerów.

Innymi słowy, kod przeprowadzający pomiary czasu może być znacznie bardziej złożony, niż można się tego było na pierwszy rzut oka spodziewać! Aby zwiększyć uniwersalność i dokładność takiego rozwiązania, rozbudujemy nasze narzędzie w nadal proste, ale znacznie bardziej użyteczne funkcje do pomiaru czasu, których będziemy mogli użyć zarówno do sprawdzenia, jak działają alternatywne opcje iteracji, jak i do zastosowania w innych scenariuszach wymagających pomiarów czasu w przyszłości. Funkcje zostaną zapisane w pliku modułu, dzięki czemu będziemy mogli ich używać w różnych programach, i będą zawierać notki dokumentacyjne z podstawowymi informacjami o sposobie użycia, które możemy wyświetlić za pomocą narzędzia PyDoc — w rozdziale 15. znajdziesz rysunek 15.2 przedstawiający zrzut ekranu ze stronami dokumentacji modułów czasowych, które tutaj tworzymy:

```
# Plik timer.py

"""
Własne narzędzia do pomiaru czasów wywołań funkcji.

Zwraca całkowity czas działania, najlepszy czas i najlepszy czas całkowity
"""

import time, sys

timer = time.clock if sys.platform[:3] == 'win' else time.time

def total(reps, func, *pargs, **kargs):

    """
    Całkowity czas wywołania funkcji func() reps razy

    Zwraca (czas całkowity, ostatni wynik)
    """

    repslist = list(range(reps))          # nie uwzględniamy czasu działania range; wyrównanie
                                         # różnic w 2.x i 3.x
    start = timer()                      # lub używamy perf_counter/innej metody w wersji 3.3+
    for i in repslist:
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
        return (elapsed, ret)

def bestof(reps, func, *pargs, **kargs):

    """
    Najlepsze wykonanie funkcji func() wśród reps wywołań

    Zwraca (najlepszy czas, ostatni wynik)
    """

    best = 2 ** 32                         # 136 lat to wystarczająco długo
    for i in range(reps):                  # czas działania range nie jest tutaj uwzględniany
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start          # lub wywołaj total() z reps=1
        if elapsed < best: best = elapsed  # lub dodaj do listy i pobierz min()
    return (best, ret)

def bestoftotal(reps1, reps2, func, *pargs, **kargs):
```

```

    """
    Najlepszy wynik całkowity:
    (najlepszy czas z reps1 wywołań spośród (całkowity czas reps2 wywołań funkcji func()))
    """

    return bestof(reps1, total, reps2, func, *pargs, **kargs)

```

Pod względem operacyjnym moduł ten mierzy zarówno *czas całkowity*, jak i *czas najszybszego* wywołania funkcji, oraz zagnieżdżony *najlepszy czas całkowity* z łącznych zestawień dwóch pozostałych. W każdym przypadku mierzy czas wywołania dowolnej funkcji z dowolną liczbą argumentów pozycyjnych i argumentów ze słowami kluczowymi, pobierając czas rozpoczęcia, wywołując badaną funkcję i odejmując czas rozpoczęcia od czasu zakończenia. Poniżej zamieszczamy krótki opis tego, w jaki sposób ta wersja rozwiązuje problem wad swojego poprzednika:

- Moduł standardowy `time` Pythona udostępnia funkcję zwracającą aktualny czas, ale jej precyza jest zależna od platformy systemowej; w systemie Windows jest to funkcja `clock`, której wywołanie zwraca wyniki z dokładnością mikrosekundową, dzięki czemu jest bardzo dokładna. Ponieważ jednak funkcja `time` w systemie Unix może być znacznie lepsza, nasz skrypt wybiera między nimi automatycznie na podstawie ciągu `platform` w module `sys`, który zaczyna się od znaków `win`, jeżeli program działa w systemie Windows. Zobacz także ramkę „Nowe metody do pomiaru czasu w wersji 3.3” w dalszej części tego rozdziału, gdzie znajdziesz informacje na temat innych opcji pomiaru czasu w wersji 3.3 i późniejszych, które nie zostały tutaj użyte dla zapewnienia pełnej przenośności kodu; będziemy również mierzyć czas w Pythonie 2.x, gdzie te nowsze rozwiązania nie są dostępne, a wyniki ich działania w systemie Windows w wersji 3.3 wydają się podobne.
- Wywołanie funkcji `range` jest wyłączone z pętli pomiarowej w funkcji `total`, dzięki czemu koszt utworzenia zakresu w wersji 2.x nie jest wliczany do pomiaru; w wersji 3.x funkcja `range` jest obiektem iterowalnym, więc ten krok nie jest ani wymagany, ani szkodliwy, ale ze względu na fakt, że nadal wynik działania jest przekształcany przez funkcję `list`, jego koszt wykonania jest taki sam zarówno w wersji 2., jak i 3.x. Nie dotyczy to funkcji `bestof`, ponieważ wywołania funkcji `range` w poiniarze czasu nie są uwzględniane.
- Liczba powtórzeń `reps` jest przekazywana jako argument przed testowaną funkcją i jej argumentami, aby umożliwić zmianę liczby powtórzeń w poszczególnych wywołaniach.
- Dowolna liczba argumentów pozycyjnych i ze słowami kluczowymi jest przetwarzana z użyciem składni `argumentów oznaczonych gwiazdką`, zatem muszą one być wysypane indywidualnie, a nie w sekwencji lub słowniku. W razie potrzeby kod wywołujący może rozpakować kolekcję argumentów do pojedynczych argumentów z gwiazdkami w wywołaniu, jak to jest robione przez funkcję `bestoftotal` na końcu modułu. Jeżeli działanie tej funkcji wydaje Ci się niezrozumiałe, powinieneś zajrzieć do rozdziału 18.
- Pierwsza funkcja w tym module zwraca krótką zawierającą *całkowity* czas wszystkich wywołań wraz z wynikiem działania badanej funkcji, dzięki czemu wywołujący może zweryfikować poprawność jej działania.
- Druga funkcja działa podobnie, ale zamiast całkowitego czasu wywołania zwraca *najlepszy* (minimalny) czas spośród wszystkich wywołań — bardzo przydatne, jeżeli chcesz szybko odfiltrować wpływ innych działań na komputerze, ale mniej użyteczne w przypadku testów, które działają zbyt szybko, aby ich czas działania był wystarczająco długi.
- Aby rozwiązać problem związany z poprzednim punktem, ostatnia funkcja w tym pliku uruchamia zagnieżdżone testy sumaryczne w ramach najlepszego testu, aby uzyskać *najlepszy czas całkowity ze wszystkich czasów całkowitych*. Zagnieżdżona suma czasów może sprawić, że testy szybko działających funkcji będą bardziej użyteczne. Kod tej funkcji może być łatwiejszy do zrozumienia, jeżeli przypomnisz sobie, że każda funkcja jest obiektem iterowalnym i dotyczy to również samych funkcji testujących.
- Dzięki temu, że funkcja mierząca wydajność została zapisana w module, staje się użytecznym narzędziem, które można zaimportować i wykorzystać w dowolnym programie. Pojęcia modułów i importowania zostały wprowadzone w rozdziale 3., a więcej szczegółowych informacji na ich temat znajdziesz w następnej części książki; na razie po prostu wystarczy zaimportować moduł i wywołać funkcję, aby użyć jednego z timerów zdefiniowanych w tym pliku. W prostych zastosowaniach moduł ten jest podobny do swojego poprzednika, ale będzie znacznie bardziej niezawodny i dokładny w bardziej złożonych kontekstach. W Pythonie 3.3 jego przykładowe działanie wygląda następująco:

```

>>> import timer
>>> timer.total(1000, pow, 2, 1000)[0]          # Porównaj z wynikami timer() z przykładu powyżej
0.0029542985410557776
>>> timer.total(1000, str.upper, 'spam')        # Zwraca krótką (czas wywołania, rezultat ostatniego
                                                # wywołania)
(0.000504845391709686, 'SPAM')
>>> timer.bestof(1000, str.upper, 'spam')        # 1/1000 całkowitego czasu działania
(4.887177027512735e-07, 'SPAM')
>>> timer.bestof(1000, pow, 2, 1000000)[0]
0.00393515497972885
>>> timer.bestof(50, timer.total, 1000, str.upper, 'spam')
(0.0005468751145372153, (0.0005004469323637295, 'SPAM'))
>>> timer.bestoftotal(50, 1000, str.upper, 'spam')

```

```
(0.000566912540591602, (0.0005195069228989269, 'SPAM'))
```

Nowe metody do pomiaru czasu w wersji 3.3

W tej sekcji wykorzystano metody `clock` i `time` modułu `time`, ponieważ dotyczą one wszystkich wersji Pythona. W wersji 3.3 wprowadzono nowe interfejsy w tym module, które zostały zaprojektowane tak, aby były bardziej przenośne. Co prawda zachowania metod `clock` i `time` tego modułu różnią się w zależności od platformy, ale jego nowe funkcje `perf_counter` i `process_time` mają dobrze zdefiniowaną i neutralną dla platformy semantykę:

- `time.perf_counter()` zwraca wyrażoną w ułamkach sekund wartość licznika wydajności, korzystającą z zegara o najwyższej dostępnej rozdzielczości do pomiaru krótkotrwalego. Obejmuje czas, który upłynął podczas uśpienia, i jest dostępna dla całego systemu.
- `time.process_time()` zwraca wartość wyrażoną w ułamkach sekund, reprezentującą sumę czasu systemu i czasu procesora bieżącego procesu użytkownika. Nie obejmuje czasu, który upłynął podczas uśpienia systemu, i z definicji obejmuje cały proces.

Dla obu tych wywołań punkt odniesienia zwracanej wartości nie jest zdefiniowany, zatem ważna jest tylko *różnica* między wynikami kolejnych wywołań. Wywołanie `perf_counter` może być używane do wyznaczania całkowitego czasu trwania zadania, a od wersji 3.3 jest domyślnie używane do pomiarów czasu w module `timeit`, o którym będziemy mówić już za chwilę; funkcja `process_time` zwraca czas procesora.

Wywołanie metody `time.clock` jest nadal możliwe do wykorzystania w systemie Windows, jak to pokazaliśmy w tej książce. W dokumentacji wersji 3.3 metoda ta jest już oznaczona jako przestarzała, ale nie wyświetla żadnego ostrzeżenia, gdy jest używana, co oznacza, że może zostać oficjalnie uznana za przestarzałą dopiero w późniejszych wersjach. W razie potrzeby możesz wykryć wersję 3.3 lub nowszą Pythona za pomocą kodu pokazanego poniżej, którego jednak postanowiłem nie używać w przykładach ze względu na chęć zachowania zwięzości i łatwości porównywania timerów:

```
if sys.version_info[0] >= 3 and sys.version_info[1] >= 3:  
    timer = time.perf_counter # lub process_time  
else:  
    timer = time.clock if sys.platform[:3] == 'win' else time.time  
  
Alternatywnym rozwiązaniem może być kod przedstawiony poniżej, który zwiększyłby również przenośność i  
odizolował Cię od metod i funkcji, które mogą się okazać przestarzałe w przyszłości; niemniej opiera się on na  
zgłaszanu wyjątków, których jeszcze nie omawialiśmy w pełni, a jego użycie może również spowodować, że  
porównania prędkości poszczególnych rozwiązań między różnymi wersjami będą niepoprawne — zegary w różnych  
wersjach i platformach mogą różnić się rozdzielczością!  
  
try:  
    timer = time.perf_counter # lub process_time  
except AttributeError:  
    timer = time.clock if sys.platform[:3] == 'win' else time.time
```

Gdybym pisał tę książkę tylko z myślą o użytkownikach Pythona 3.3+, użylibym tutaj nowych i znacznie ulepszonych wywołań, których również powinieneś używać w swojej pracy, jeżeli korzystasz wyłącznie z nowych wersji. Nowsze wywołania nie będą jednak działać dla użytkowników starszych wersji, a nadal jest to większość dzisiejszego „pythonowego” świata. Oczywiście łatwiej byłoby udawać, że przeszłość nie ma znaczenia, tyle że byłoby to nie tylko zaklinaniem rzeczywistości, ale również mogłoby to być po prostu niegrzeczne.

Ostatnie dwa wywołania powyżej obliczają *najlepsze czasy całkowite* — najniższy czas spośród 50 przebiegów timera, z których każdy oblicza całkowity czas wywołania funkcji `str.upper` 1000 razy (mniej więcej odpowiada to całkowitemu czasowi 1000-krotnego wywołania tej funkcji na początku przykładu). Funkcja użyta w ostatnim wywołaniu jest tak naprawdę tylko wygodnym ułatwieniem, które odwzorowuje poprzedni sposób wywołania; oba zwracają krotkę będącą „najlepszą z wynikowych krotek”, zawierającą krotkę z ostatnim, całkowitym czasem wykonania i wynikiem działania funkcji.

Porównaj te dwa ostatnie wyniki z następującą alternatywną opartą na generatorze:

```
>>> min(timer.total(1000, str.upper, 'spam') for i in range(50))  
(0.0005155971812769167, 'SPAM')
```

Wyznaczając w ten sposób minimalny czas całkowity, otrzymujemy podobny efekt, ponieważ czasy w krotkach wynikowych dominują w porównaniach wykonanych przez `min` (znajdują się w krotkach najbardziej z lewej strony). Możemy tego użyć również w naszym module (i zrobimy tak w późniejszych wersjach); wyniki działania timerów zmieniają się nieznacznie dzięki pominięciu bardzo małego narzutu na kod wyznaczający najlepszy czas i braku zagieźdzania krotek wyników, chociaż oba rezultaty wystarczają do przeprowadzenia porównań względnych. W tej wersji do poprawnego działania funkcja musi wybrać wysoką wartość początkową najlepszego czasu całkowitego — choć zastosowane tutaj 136 lat jest najprawdopodobniej znacznie dłuższym okresem czasu niż w zdecydowanej większości testów, które kiedykolwiek będziesz w stanie przeprowadzić!

```
>>> (((2 ** 32) / 60) / 60) / 24) / 365 # 136 lat, dwa miesiące i kilka dni  
136.19251953323186
```

```
>>> (((2 ** 32) // 60) // 60) // 24) // 365          # Funkcja floor(): zobacz rozdział 5.  
136
```

Skrypt mierzący wydajność

Aby zmierzyć wydajność narzędzia iteracyjnego (co było i jest od początku naszym głównym celem), wystarczy uruchomić skrypt prezentowany na poniższym listingu. Skrypt wykorzystuje moduł `timer` napisany wcześniej i dokonuje pomiarów czasu działania narzędzi tworzących listy, które poznaliśmy dotychczas.

```
# Plik timeseqs.py  
"Mierzy względną szybkość działania różnych narzędzi iteracyjnych."  
import sys, timer                                         # Import funkcji pomiaru czasu  
reps = 10000  
repslist = list(range(reps))                             # nie uwzględniamy czasu działania range; wyrównanie  
                                                       # różnic w 2.x i 3.x  
def forLoop():  
    res = []  
    for x in repslist:  
        res.append(abs(x))  
    return res  
def listComp(): return [abs(x) for x in repslist]  
def mapCall():  
    return list(map(abs, repslist))                         # wywołania list() używamy tutaj tylko w wersji 3.x!  
# return map(abs, repslist)  
def genExpr():  
    return list(abs(x) for x in repslist)      # wywołanie list() niezbędne do zwrócenia wszystkich  
                                               # wyników  
def genFunc(): def gen():  
    for x in repslist:  
        yield abs(x)  
    return list(gen())                                    # wywołanie list() niezbędne do zwrócenia  
wszystkich wyników  
print(sys.version) for test in (forLoop, listComp, mapCall, genExpr, genFunc):  
    (bestof, (total, result)) = timer.bestoftotal(5, 1000, test)  
    print ('%-9s: %.5f => [%s...%s]' %  
          (test.__name__, bestof, result[0], result[-1]))
```

Skrypt testuje pięć alternatywnych sposobów tworzenia list wyników. Jak pokazano w przykładzie, podane czasy odzwierciedlają 10 milionów wywołań dla każdej z pięciu testowanych funkcji — każda z nich tworzy listę 10 000 pozycji po 1000 razy. Ten proces powtarza się 5 razy, aby uzyskać jak najbardziej miarodajny czas dla każdej z 5 testowanych funkcji, dając w sumie aż 250 milionów kroków dla całego skryptu (wartość dosyć imponująca, ale całkiem rozsądna na większości współczesnych komputerów).

Zwrócić uwagę na to, że wyniki wyrażeń generatorów są przekształcane na listę za pomocą wbudowanej funkcji `list`, aby zmusić je do utworzenia wszystkich swoich wartości; jeżeli byśmy tego nie zrobili, zarówno w wersji 2.x, jak i 3.x utworzylibyśmy proste generatorы, które nigdy nie wykonują żadnej prawdziwej pracy. W Pythonie 3.x musimy zrobić to samo dla wyniku działania funkcji `map`, ponieważ jest to również obiekt iterowalny; w wersji 2.x wywołanie funkcji `list` dla wyników funkcji `map` musi jednak zostać usunięte ręcznie, aby uniknąć naliczania dodatkowego czasu związanego z tworzeniem listy (choć w większości testów wpływ takiego wywołania wydaje się być znikomy).

W podobny sposób wynik działania funkcji `range` z wewnętrznych pętli jest przenoszony na góre modułu, tak aby usunąć narzuć związanego z jej działaniem z czasu całkowitego, i opakowany w wywołanie funkcji `list`, aby jego koszt ogólny nie był zniekształcony przez bycie generatorem tylko w wersji 3.x (podobnie jak to zrobiliśmy w module `timer`). Ogólny rezultat może być nieco zniekształcony przez koszt wykonywania wewnętrznej pętli iteracji, ale mimo to najlepszym rozwiązaniem jest usunięcie z równania jak największej liczby zmiennych.

Zwróć również uwagę na sposób, w jaki kod przechodzi przez krotkę pięciu obiektów funkcji i wypisuje wartość atrybutu `__name__` każdego z nich: jak widzieliśmy wcześniej, jest to wbudowany atrybut, przechowujący nazwę funkcji^[1].

Wyniki pomiarów czasu

Gdy przedstawiony wyżej skrypt uruchomilem w *Pythonie 3.3* na moim laptopie z systemem Windows 7, okazało się, że funkcja `map` jest nieco szybsza od list składanych, obie te metody są szybsze od pętli `for`, a wyrażenia generatorów i funkcje plasują się w środku stawki (całkowite czasy wykonania, pokazane poniżej, są wyrażone w sekundach):

```
C:\code> c:\python33\python timeseqs.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
forLoop : 1.33290 => [0...9999]
listComp : 0.69658 => [0...9999]
mapCall : 0.56483 => [0...9999]
genExpr : 1.08457 => [0...9999]
genFunc : 1.07623 => [0...9999]
```

Gdy wystarczająco uważnie przyjrzymy się tym wynikom i kodowi skryptu pomiarowego, zauważymy, że obecnie wyrażenia generatorów działają wolniej od list składanych. Choć przekształcenie wyrażenia generatorów w listę za pomocą wywołania funkcji `list` powoduje, że staje się ono *funkcjonalnym* odpowiadniukiem listy składanej, to *wewnętrzna implementacja* każdej z tych technologii jest na tyle odmienna, że występują różnice w wydajności działania tych dwóch rodzajów wyrażeń (należy jednak pamiętać, że w przypadku testu generatora do wyniku również jest doliczane wywołanie funkcji `list`).

```
return [abs(x) for x in repstlist]           # 0,69 sekundy
return list(abs(x) for x in repstlist)         # 1,08 sekundy: wewnętrzne różnice implementacji
```

Chociaż dokładne wyjaśnienie przyczyny takiego zachowania wymagałoby głębszego rozważenia (i być może szczegółowej analizy kodu źródłowego Pythona), wydaje się to mieć sens, biorąc pod uwagę, że wyrażenie generatora musi wykonać dodatkową pracę, aby zapisać i przywrócić swój stan podczas tworzenia wartości; listy złożone tego robić nie muszą i w związku z tym działają nieco szybciej zarówno tutaj, jak i w późniejszych testach.

Co ciekawe, kiedy uruchamiałem ten sam test w Pythonie 3.0 w systemie Windows Vista podczas pracy nad czwartą edycją tej książki oraz w systemie Windows XP z Pythonem 2.5 przy pracy nad trzecią edycją, wyniki były dosyć podobne — listy składane okazały się być prawie dwukrotnie wydajniejsze od pętli `for`, zaś funkcja `map` była nieco szybsza od list składanych w przypadku użycia funkcji wbudowanych, jak `abs` (funkcja obliczająca wartość bezwzględną). Bezwzględne czasy Pythona 2.5 były około czterej do pięciu razy wolniejsze niż obecne wyniki wersji 3.3, ale prawdopodobnie znacznie szybsze współczesne komputery mają na to o wiele większy wpływ niż jakiekolwiek ulepszenia dokonane w Pythonie.

W rzeczywistości większość wyników działania tego skryptu na tej samej maszynie w *Pythonie 2.7* jest obecnie nieco szybsza niż w wersji 3.3 — usunąłem jedynie wywołanie funkcji `list` na wynikach funkcji `map`, aby uniknąć zdublowanego tworzenia listy wyników, choć wpływ pozostawienia tego wywołania w kodzie jest stałym i naprawdę niewielkim:

```
c:\code> c:\python27\python timeseqs.py
2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)]
forLoop : 1.24902 => [0...9999]
listComp : 0.66970 => [0...9999]
mapCall : 0.57018 => [0...9999]
genExpr : 0.90339 => [0...9999]
genFunc : 0.90542 => [0...9999]
```

Dla porównania poniżej przedstawiono wyniki tych samych testów prędkości w dystrybucji *PyPy*, zoptymalizowanej implementacji Pythona omówionej w rozdziale 2., której bieżąca wersja 1.9 implementuje język Python 2.7. *PyPy* jest w przybliżeniu 10 razy szybszy (rzad wielkości!); a wypada jeszcze lepiej, gdy ponownie przejdziemy do porównania wersji Pythona w dalszej części tego rozdziału, używając narzędzi o różnych strukturach kodu (choć traci nieco w kilku innych testach):

```
c:\code> c:\PyPy\pypy-1.9\pypy.exe timeseqs.py
2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit]
forLoop : 0.10106 => [0...9999]
listComp : 0.05629 => [0...9999]
mapCall : 0.10022 => [0...9999]
```

```
genExpr  : 0.17234 => [0...9999]
genFunc  : 0.17519 => [0...9999]
```

W dystrybucji PyPy w naszym teście listy składane pokonują funkcję `map`, ale fakt, że wszystkie wyniki działania PyPy są znacznie szybsze, wydaje się być bardziej istotny. W dystrybucjach CPython funkcja `map` jest jak dotąd najszybsza.

Wpływ wywołań funkcji: `map`

Zobaczmy jednak, co się stanie, jeżeli zmienimy ten skrypt tak, aby na każdej iteracji wykonywał operację wbudowaną, taką jak dodawanie, zamiast wywoływać funkcję wbudowaną, np. `abs` (pominięte fragmenty poniższego pliku są takie same jak poprzednio, a wyniki funkcji `map` zostały opakowane w wywołanie funkcji `list` w celu przetestowania tylko w wersji 3.3):

```
# Plik timeseqs2.py (wiersze różnicowe)...

...
def forLoop():
    res = []
    for x in repslist:
        res.append(x + 10)
    return res

def listComp(): return [x + 10 for x in repslist]

def mapCall():
    return list(map((lambda x: x + 10), repslist))      # wywołanie list() tylko dla wersji 3.x

def genExpr():
    return list(x + 10 for x in repslist)                # wywołanie list() dla wersji 2.x + 3.x

def genFunc(): def gen():
    for x in repslist:
        yield x + 10
    return list(gen())                                    # wywołanie list() dla wersji 2.x + 3.x

...

```

W tym przypadku konieczność wywołania funkcji zdefiniowanej przez użytkownika w funkcji `map` powoduje, że ta metoda staje się wolniejsza od pętli `for`, mimo że pętla wymaga większej ilości kodu — lub równoważnie, usunięcie wywołań funkcji może spowodować, że pozostałe wywołania działają szybciej (więcej na ten temat już niebawem). W Pythonie 3.3 wyglądało to następująco:

```
c:\code> c:\python33\python timeseqs2.py
3.3.0 (v3.3.0:bd8af90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
forLoop  : 1.35136 => [10...10009]
listComp : 0.73730 => [10...10009]
mapCall  : 1.68588 => [10...10009]
genExpr  : 1.10963 => [10...10009]
genFunc  : 1.11074 => [10...10009]
```

Wyniki te były również spójne w dystrybucji CPython. Rezultaty z poprzedniej edycji w wersji 3.0 na wolniejszej maszynie ponownie były dosyć podobne, choć około dwa razy wolniejsze z powodu różnic między maszynami używanymi do przeprowadzenia testów (wyniki z wersji 2.5 Pythona na jeszcze wolniejszej maszynie były mniej więcej cztery do pięciu razy wolniejsze niż najnowsze wyniki).

Z powodu znaczących wewnętrznych optymalizacji interpretera tego typu porównania wydajności algorytmów są zadaniem dość niewdzięcznym. Bez twardych danych liczbowych w zasadzie niemożliwe jest odgadnięcie, która metoda okaże się szybsza. Najlepszym rozwiązaniem jest zaimplementowanie interesujących Cię rozwiązań i dokonanie pomiarów na komputerze, na którym kod będzie wykonywany, z taką wersją Pythona, z jaką będzie pracował na co dzień.

W naszym przypadku możemy z całą pewnością powiedzieć, że w tej wersji Pythona użycie funkcji zdefiniowanej przez użytkownika w wywołaniach funkcji `map` wydaje się znacznie spowalniać działanie (choć operator `+` również może być wolniejszy niż trywialna funkcja `abs`), a listy składane działają tutaj najszybciej (choć są wolniejsze niż funkcja `map` w niektórych innych rozwiązaniach). Listy składane wydają się konsekwentnie być dwa razy szybsze niż pętle `for`, ale nawet to musi zostać odpowiednio zakwalifikowane — na względną szybkość działania list składanych może wpływać ich dodatkowa składnia (np. klauzule `if`), zmiany w wersjach Pythona i sposoby używania, o których tutaj nie wspominaliśmy.

Jednak jak wspominaliśmy wcześniej, wydajność kodu nie powinna być najistotniejszym czynnikiem w pracy z Pythonem. Pierwszą rzeczą, jaką powinniśmy zrobić, optymalizując kod w Pythonie, jest... nie optymalizować tego kodu! Kod piszemy z myślą o czytelności i prostocie, następnie go optymalizujemy, ale tylko wówczas, gdy wystąpi taka konieczność. Może się bowiem okazać, że wydajność każdej z pięciu przedstawionych alternatyw jest w zupełności wystarczająca dla przetwarzania danych, którymi będzie zajmował się program, a w takiej sytuacji zawsze powinieneś wybrać rozwiązanie najkorzystniejsze z punktu widzenia czytelności programu.



Aby dowiedzieć się czegoś więcej, spróbuj zmienić kod tak, aby zastosować prostą funkcję zdefiniowaną przez użytkownika we wszystkich pięciu technikach iteracji. Na przykład (poniższy kod znajdziesz w pliku *timeseqs2B.py* w pakietie przykładów do książki):

```
def F(x): return x
def listComp():
    return [F(x) for x in repslist]
def mapCall():
    return list(map(F, repslist))
```

Wyniki, zapisane w przykładowym pliku *timeseqs-results.txt*, są względnie podobne do wyników użycia wbudowanych funkcji, takich jak `abs` — a przynajmniej w dystrybucji CPython funkcja `map` jest najszybsza. Ogólnie rzecz biorąc, spośród pięciu przedstawionych technik iteracji funkcja `map` jest obecnie najszybsza, jeżeli wszystkie pięć rozwiązań wywołuje dowolną funkcję, wbudowaną lub nie, ale najwolniejsza, gdy inne formy iteracji tego nie robią.

Oznacza to, że funkcja `map` wydaje się być wolniejsza po prostu dlatego, że *wymaga wywołań funkcji*, a wywołania funkcji są ogólnie stosunkowo wolne. Ponieważ funkcja `map` nie może uniknąć wywoływanego innego funkcji, może po prostu tracić na szybkości przez swoją specyfikę! Inne narzędzia iteracyjne wygrywają, ponieważ mogą działać bez wywoływanego funkcji. Udowodnimy to odkrycie w testach przeprowadzanych w ramach modułu `timeit`, które już są przed nami.

Inne rozwiązania dla modułu do pomiaru czasu

Moduł do pomiaru czasu zaprezentowany w poprzedniej sekcji działa, ale mógłby być nieco bardziej przyjazny dla użytkownika. Przede wszystkim jego funkcje wymagają przekazywania licznika powtórzeń jako pierwszego argumentu i nie dostarczają żadnych domyślnych wartości — być może jest to drobiazg, ale z pewnością nie jest to idealne rozwiązanie w narzędziach ogólnego przeznaczenia. Możemy również wykorzystać technikę z funkcją `min`, którą widzieliśmy już wcześniej, aby nieco uprościć sposób zwracania wartości i jeszcze bardziej zminimalizować ogólne obciążenie funkcji.

Poniżej przedstawiamy alternatywną, bardziej zaawansowaną wersję modułu do pomiaru czasu, który odnosi się do powyższych zastrzeżeń, umożliwiając przekazanie liczby powtórzeń jako argumentu ze słowem kluczowym o nazwie `_reps`:

```
# Plik timer2.py (2.x i 3.x)
"""
total(spam, 1, 2, a=3, b=4, _reps=1000) mierzy czas działania funkcji spam(1, 2, a=3, b=4)
wywołując ją _reps razy, i zwraca całkowity czas działania oraz wynik ostatniego wywołania
bestof(spam, 1, 2, a=3, b=4, _reps=5) wywołuje funkcję spam(1, 2, a=3, b=4) _reps razy
i zwraca najlepszy czas wykonania, co pozwala odfiltrować wariancję pomiaru czasu spowodowane
zmianami w bieżącym obciążeniu systemu
bestoftotal(spam, 1, 2, a=3, b=4, _reps1=5, _reps=1000) wykonuje pomiar całkowitego czasu działania
funkcji spam(1, 2, a=3, b=4) _reps1 razy, wywołując ją w każdym przebiegu po _reps2 razy i zwracając
najlepszy całkowity czas wykonania ze wszystkich _reps1 przebiegów.
"""

import time, sys
timer = time.clock if sys.platform[:3] == 'win' else time.time
def total(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 1000)                      # Przekazana lub domyślna liczba wywołań
    repslist = list(range(_reps))                          # Nie uwzględniamy czasu range dla wersji 2.x
    start = timer()
    for i in repslist:
        ret = func(*pargs, **kargs)
```

```
elapsed = timer() - start
return (elapsed, ret)

def bestof(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 5)
    best = 2 ** 32
    for i in range(_reps):
        start = timer()
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
        if elapsed < best: best = elapsed
    return (best, ret)

def bestoftotal(func, *pargs, **kargs):
    _reps1 = kargs.pop('_reps1', 5)
    return min(total(func, *pargs, **kargs) for i in range(_reps1))
```

Notka dokumentacyjna modułu prezentuje sposób jego użycia. Moduł wykorzystuje metodę `pop` słownika, aby usunąć argument `_reps` z listy argumentów przeznaczonych dla testowanej funkcji i nadać mu wartość domyślną (ma nietypową nazwę, aby uniknąć kolizji z prawdziwymi argumentami ze słowami kluczowymi, przeznaczonymi dla testowanej funkcji).

Zwróć uwagę, w jaki sposób mechanizmy wyznaczania najlepszego czasu wykonania używają funkcji `min` i generatora, który widzieliśmy wcześniej, zamiast wywołań zagnieźdzonych, po części dlatego, że upraszcza to wyniki i pozwala uniknąć niewielkiego, dodatkowego narzutu czasu z poprzedniej wersji (gdzie czas najlepszego wykonania pobierany był *po wyznaczeniu* całkowitego czasu działania), ale także dlatego, że musi obsługiwać dwa różne słowa kluczowe reprezentujące liczby powtarzane z wartościami domyślnymi – funkcje `total` i `bestof` nie mogą używać tej samej nazwy argumentu. Jeżeli chcesz sobie ułatwić śledzenie sposobu działania tych funkcji, możesz w odpowiednich miejscach kodu wstawić polecenia wyświetlające na ekranie wartości wybranych zmiennych i argumentów.

Aby przetestować ten nowy moduł do pomiaru czasu, możesz zmienić skrypty w sposób pokazany poniżej lub użyć kodu zapisanego w pliku `timeseqs_timer2.py` w pakiecie przykładów do książki; wyniki działania powinny być zasadniczo takie same jak wcześniej (jest to przede wszystkim zmiana interfejsu API), więc nie będziemy ich tutaj ponownie pokazywać:

```
import sys, timer2
...
for test in (forLoop, listComp, mapCall, genExpr, genFunc):
    (total, result) = timer2.bestoftotal(test, _reps1=5, _reps=1000)
# Lub:
#     (total, result) = timer2.bestoftotal(test)
#     (total, result) = timer2.bestof(test, _reps=5)
#     (total, result) = timer2.total(test, _reps=1000)
#     (bestof, (total, result)) = timer2.bestof(timer2.total, test, _reps=5)
    print ('%-9s: %.5f => [%s...%s]' %
          (test.name, total, result[0], result[-1]))
```

Możesz także przeprowadzić kilka interaktywnych testów, tak jak to zrobiliśmy w oryginalnej wersji — wyniki są bardzo zbliżone do tych uzyskanych poprzednio, ale przekazujemy liczbę powtórzeń jako argumenty ze słowami kluczowymi, które jeżeli zostaną pominięte, dostarczają wartości domyślnych; w Pythonie 3.3 wyglądało to następująco:

```

0.0007550688578703557
>>> bestof(pow, 2, 1000000, _reps=30)[0]           # 2 ** 1M, najlepszy czas z 30 powtórzeń
0.004040229286800923
>>> bestoftotal(str.upper, 'spam', _reps1=30, _reps=1000)   # najlepszy czas z 30 przebiegów

# po 1K powtórzeń
(0.0004945823198454491, 'SPAM')
>>> bestof(total, str.upper, 'spam', _reps=30)           # Wywołania zagnieżdżone również działają
(0.0005463863968202531, (0.0004994694969298052, 'SPAM'))

```

Aby przekonać się, w jaki sposób obsługiwane są teraz słowa kluczowe, zdefiniuj funkcję z większą liczbą argumentów i przekaź część z nich po nazwie:

```

>>> def spam(a, b, c, d): return a + b + c + d
>>> total(spam, 1, 2, c=3, d=4, _reps=1000)
(0.0009730369554290519, 10)
>>> bestof(spam, 1, 2, c=3, d=4, _reps=1000)
(9.774353202374186e-07, 10)
>>> bestoftotal(spam, 1, 2, c=3, d=4, _reps1=1000, _reps=1000)
(0.00037289161070930277, 10)
>>> bestoftotal(spam, *(1, 2), _reps1=1000, _reps=1000, **dict(c=3, d=4))
(0.00037289161070930277, 10)

```

Użycie argumentów ze słowami kluczowymi w wersji 3.x

Nadszedł czas na ostatni punkt w tym wątku: w celu uproszczenia kodu modułu liczników czasu możemy skorzystać z *argumentów ze słowami kluczowymi*, dostępnych w Pythonie 3.x. Jak się dowiedzieliśmy w rozdziale 18., argumenty takie są doskonałym rozwiązaniem w sytuacji, gdy potrzebujemy dodatkowego argumentu, takiego jak `_reps`. Argumenty te muszą być zadeklarowane w nagłówku funkcji po argumentach pozycyjnych `*`, a przed argumentami ze słowami kluczowymi `**`, zaś w wywołaniu funkcji muszą być przekazane przez klucze i muszą występować przed argumentem `**`, o ile został użyty. Poniższy listing prezentuje kod będący modyfikacją poprzedniego i wykorzystujący argument ze słowem kluczowym. Choć ten kod jest prostszy, można go poprawnie skompilować i uruchomić wyłącznie w wersji 3.x i nie będzie działał w 2.x.

```

# plik timer3.py (tylko dla wersji 3.x)
"""

Taki sam sposób użycia jak modułu timer2.py, ale ta wersja używa prostszych argumentów
ze słowami kluczowymi i wartościami domylnymi, dostępnych tylko w wersji 3.x. Nie ma tutaj potrzeby
przenoszenia wywołania range () poza pętlę pomiaru, ponieważ w wersji 3.x jest zawsze generatorem;
takie rozwiązanie nie będzie działać w wersji 2.x.

"""

import time, sys
timer = time.clock if sys.platform[:3] == 'win' else time.time
def total(func, *pargs, _reps=1000, **kargs):
    start = timer()
    for i in range(_reps):
        ret = func(*pargs, **kargs)
        elapsed = timer() - start
    return (elapsed, ret)
def bestof(func, *pargs, _reps=5, **kargs):
    best = 2 ** 32
    for i in range(_reps):
        start = timer()

```

```

    ret = func(*pargs, **kargs)
    elapsed = timer() - start
    if elapsed < best: best = elapsed
    return (best, ret)
def bestoftotal(func, *pargs, _reps=5, **kargs):
    return min(total(func, *pargs, **kargs) for i in range(_reps))

```

Tej wersji używamy w taki sam sposób jak poprzedniej i daje ona identyczne wyniki, więc nie będziemy ponownie przytaczać takich samych wyników opartych na tych samych testach; jeżeli chcesz, możesz tutaj poeksperymentować samodzielnie. Jeżeli jednak się na to zdecydujesz, zwróć uwagę na reguły kolejności argumentów w wywołaniach funkcji. Na przykład poprzednia funkcja `total` była wywoływana w następujący sposób:

```
(elapsed, ret) = total(func, *pargs, _reps=1, **kargs)
```

Więcej informacji na temat argumentów ze słowami kluczowymi w wersji 3.x znajdziesz w rozdziale 18.; mogą one zdecydowanie uprościć kod dla wielu konfigurowalnych narzędzi takich jak pokazane powyżej, ale nie są wstecznie kompatybilne z Pythonem 2.x. Jeżeli chcesz porównać szybkość działania Pythona 2.x i 3.x lub wesprzeć programistów korzystających z różnych wersji Pythona, poprzednia wersja modułu będzie najprawdopodobniej lepszym wyborem.

Powinieneś również pamiętać, że w przypadku trywialnych funkcji, takich jak niektóre testowane w poprzedniej wersji modułu, koszty działania kodu samego timera mogą być bardzo zbliżone do kosztów działania testowanej funkcji, więc w takich scenariuszach nie powinieneś brać zmierzonych czasów działania zbyt dosłownie. Wyniki pomiarów mogą jednak pomóc w ocenie względnej prędkości alternatywnych rozwiązań kodowania i mogą mieć większe znaczenie dla operacji, które trwają dłużej lub są często powtarzane.

Inne sugestie

Dalsze eksperymenty w dziedzinie pomiaru czasu wykonania kodu mogą polegać na próbach modyfikowania liczby powtórzeń wykorzystywanych przez moduły. Warto też przetestować standardowy moduł `timeit`, który automatyzuje synchronizację kodu, obsługuje tryby użycia wiersza poleceń i rozwiązuje niektóre problemy specyficzne dla platformy — więcej szczegółowych informacji na ten temat znajdziesz w następnym podrozdziale.

Mozesz także zajrzieć do standardowego modułu biblioteki `profile`, będącej kompletnym narzędziem do profilowania kodu źródłowego. Więcej na jego temat dowiemy się w rozdziale 36. w kontekście tworzenia narzędzi programistycznych na potrzeby dużych, złożonych projektów. Ogólnie rzecz biorąc, profilowanie kodu pozwala na zidentyfikowanie i wyodrębnienie wąskich gardel programu przed rozpoczęciem modyfikowania kodu i przeprowadzania testów wydajności, jakie robiliśmy w tym rozdziale.

Mozesz użyć odpowiednio zmodyfikowanych wersji skryptów mierzących szybkość działania do sprawdzenia wydajności *zbiorów i słowników składanych* w wersjach 3.x i 2.7, przedstawionych w poprzednim rozdziale, oraz ich odpowiedników z użyciem pętli `for`. Są one znacznie mniej powszechnie spotykane w programach Pythona niż rozwiązania wykorzystujące budowanie list wyników, więc ich przetestowanie pozostawimy jako sugerowane ćwiczenie do samodzielnego wykonania, choć następna sekcja częściowo zepsuje nam całą niespodziankę.

Na koniec zapisz i zachowaj utworzony tutaj moduł pomiarów czasu do wykorzystania w przyszłości — niebawem zmienimy jego przeznaczenie na pomiary wydajności alternatywnych sposobów obliczania wartości pierwiastków kwadratowych w ćwiczeniach na końcu tego rozdziału. Jeżeli chcesz dalej zajmować się tym tematem, kilka propozycji do samodzielnego poeksperymentowania z technikami badania szybkości działania słowników składanych i porównywania ich do odpowiedników z pętlami `for` znajdziesz w ćwiczeniach na końcu rozdziału.

Mierzenie czasu iteracji z wykorzystaniem modułu `timeit`

W poprzedniej sekcji wykorzystaliśmy nasze własne funkcje czasowe do porównania szybkości działania kodu. Jak już wspominaliśmy wcześniej, standardowa biblioteka Pythona jest dostarczana z modulem o nazwie `timeit`, którego można używać w podobny sposób, ale który oferuje dodatkową elastyczność i może lepiej izolować klientów od pewnych różnic w działaniu kodu na różnych platformach.

Jak zwykle w Pythonie, ważne jest zrozumienie podstawowych reguł i zasad, takich jak te przedstawione w poprzednim rozdziale. Powszechnie w Pythonie podejście typu „baterie dołączone do zestawu” oznacza, że zwykle w danym module znajdziesz wszystkie narzędzia, które będą Ci potrzebne, choć i tak musisz znać koncepcje leżące u podstaw ich działania, aby właściwie z nich korzystać. Rzeczywiście, moduł `timeit` jest tego najlepszym przykładem — bardzo często zdarzało się, że był niewłaściwie używany przez ludzi, którzy zupełnie nie rozumieli zasad jego działania. Teraz gdy poznaliśmy podstawy zagadnienia związane z pomiarami czasu, przejdziemy do omawiania narzędzia, które może zautomatyzować większość naszej pracy.

Podstawowe reguły korzystania z modułu timeit

Zacznijmy od omówienia podstawowych zagadnień związanych z modelem `timeit`, zanim zaczniemy wykorzystywać go w większych skryptach. W module `timeit` testy są określane przez *obiekty wywoływalne* lub *ciągi instrukcji*; te ostatnie mogą zawierać wiele powiązanych ze sobą instrukcji, rozdzielonych separatorami lub znakami nowego wiersza `\n` oraz używających spacji lub tabulatorów do tworzenia wcięć instrukcji w zagnieżdżonych blokach (np. `\n\t`). Testy mogą również obejmować instalacje i mogą być uruchamiane zarówno z poziomu wiersza poleceń, jak i wywołań interfejsu API, a także z poziomu skryptów i sesji interaktywnych.

Interaktywne użycie i wywołania API

Na przykład wywołanie funkcji `repeat` modułu `timeit` zwraca listę zawierającą całkowity czas potrzebny na przeprowadzenie testu `number` razy dla każdego z `repeat` przebiegów — wynik działania funkcji `min` na tej liście daje najlepszy czas spośród wszystkich przebiegów i pomaga odfiltrować wahania obciążenia systemu, które w przeciwnym razie mogłyby znieksztalcać wyniki pomiarów.

Poniżej pokazano takie wywołanie w działaniu, mierzące czas działania interpretacji listy składanej w dwóch wersjach dystrybucji *CPython* i zoptymalizowanej implementacji *PyPy*, którą opisywaliśmy w rozdziale 2. (obecnie obsługuje ona kod Pythona 2.7). Przedstawione tutaj wyniki dają najlepszy całkowity czas w sekundach spośród 5 przebiegów, z których każdy wykonuje blok kodu 1000 razy; sam blok kodu tworzy za każdym razem listę zawierającą 1000 elementów całkowitych (więcej szczegółowych informacji na temat programu uruchomieniowego (launcher) Pythona, używanego w pierwszych dwóch wywołaniach, znajdziesz w dodatku B).

```
c:\code> py -3
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit...
>>> import timeit
>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.5062382371756811

c:\code> py -2
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] on win32
>>> import timeit
>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.0708020004193198

c:\code> c:\pypy\pypy-1.9\pypy.exe
Python 2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit] on win32
>>>> import timeit
>>>> min(timeit.repeat(stmt="[x ** 2 for x in range(1000)]", number=1000, repeat=5))
0.005930329674303905
```

Z pewnością zauważyleś, że *PyPy* działa tutaj 10 razy szybciej niż *CPython* 2.7, a nawet 100 razy szybciej niż *CPython* 3.3, mimo że *PyPy* jest potencjalnie wolniejszym, 32-bitowym kompilatorem. Jest to oczywiście niewielki, sztuczny test porównawczy, ale jego wyniki są dosyć oszałamiające i odzwierciedlają względny ranking prędkości poszczególnych wersji, który generalnie sprawdza się również w innych testach przeprowadzanych w tej książce (choć jak zobaczymy niebawem, w niektórych zastosowaniach *CPython* wciąż potrafi pokonać dystrybucję *PyPy*).

Ten konkretny test mierzy szybkość zarówno działania listy składanej, jak i matematyki liczb całkowitych. Ta ostatnia różni się w zależności od wersji: *CPython* 3.x ma jeden rodzaj liczb całkowitych, a *CPython* 2.x ma zarówno zwykłe liczby całkowite (`int`), jak i długie liczby całkowite (`long`). Może to częściowo wyjaśniać wielkość różnic, co nie zmienia faktu, że wyniki są jednak prawdziwe i całkowicie jednoznaczne. Testy na liczbach niecałkowitych dają podobne rezultaty (np. test zmiennoprzecinkowy w rozwiązańach do ćwiczeń z tej części), a matematyka liczb całkowitych ma tutaj znaczenie — przyspieszenie działania o jeden czy dwa rzędy wielkości może być bardzo istotne w wielu prawdziwych programach, ponieważ operacje na liczbach całkowitych oraz iteracje są wszechobecne w kodzie Pythona.

Przedstawione wyniki różnią się również od względnych prędkości poszczególnych wersji z poprzedniej sekcji, w których *CPython* 2.7 był nieco szybszy niż 3.3, a *PyPy* był dziesięciokrotnie szybszy, co potwierdza większość innych testów w tej książce. Oprócz różnych rozwiązań kodu do pomiaru czasu duży wpływ na otrzymane wyniki może mieć również inną strukturę kodowania wewnętrz modułu `timeit` — w przypadku bloków kodu, takich jak testowane tutaj, `timeit` buduje, kompliuje i wykonuje ciąg instrukcji funkcji `def`, w którym osadza testowany blok kodu, tym samym unikając wywołania funkcji wewnętrznej pętli. Jak jednak zobaczymy w następnym rozdziale, wydaje się to być zupełnie nieistotne z perspektywy względnej szybkości działania.

Korzystanie z poziomu wiersza polecenia

Moduł `timeit` ma rozsądne ustawienia domyślne i może być również uruchamiany jako skrypt, albo przez podanie pełnej ścieżki i nazwy pliku, albo wywołanie z flagą `-m` automatycznie wyszukującą plik w ścieżce wyszukiwania modułów Pythona

(patrz dodatek A). Wszystkie poniższe wywołania uruchamiają Pythona 3.3. W tym trybie `timeit` wyświetla średni czas działania *pojedynczej* pętli (-n), wyrażony w mikrosekundach (oznaczonych jako `usec`), milisekundach (`msec`) lub sekundach (`sec`); aby porównać wyniki z wartościami czasu całkowitego uzyskanyimi w innych testach, powinieneś pomnożyć je przez liczbę uruchomionych pętli — w naszym przypadku 500 `usec` × 1000 pętli daje łącznie 500 milisekund, czyli inaczej mówiąc, pół sekundy:

```
c:\code> C:\python33\Lib\timeit.py -n 1000 "[x ** 2 for x in range(1000)]"
1000 loops, best of 3: 506 usec per loop
c:\code> python -m timeit -n 1000 "[x ** 2 for x in range(1000)]"
1000 loops, best of 3: 504 usec per loop
c:\code> py -3 -m timeit -n 1000 -r 5 "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 505 usec per loop
```

W ramach przykładu możemy użyć wiersza poleceń, aby sprawdzić, czy wybór sposobu wywołania timera nie wpływa na porównania prędkości między różnymi wersjami, tak jak to robiliśmy do tej pory w tym rozdziale — wersja 3.3 domyślnie korzysta z nowych wywołań, i może to mieć znaczenie, jeżeli precyza timera będzie się znacznie różnić. Aby udowodnić, że jest to nieistotne, w przykładach poniżej używamy flagi `-c`, aby zmusić moduł `timeit` do użycia metody `time.clock` we wszystkich wersjach; jest to opcja, którą podręczniki wersji 3.3 nazywają nieaktualną, ale jest wymagana do wyrównania wyników w poprzednich wersjach (dla zachowania związkowości pokazanych niżej poleceń dodałem katalog PyPy do ścieżki systemowej mojego komputera):

```
c:\code> set PATH=%PATH%;C:\pypy\pypy-1.9
c:\code> py -3 -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 502 usec per loop
c:\code> py -2 -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 70.6 usec per loop
c:\code> pypy -m timeit -n 1000 -r 5 -c "[x ** 2 for x in range(1000)]"
1000 loops, best of 5: 5.44 usec per loop
C:\code> py -3 -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"
1000 loops, best of 5: 815 usec per loop
C:\code> py -2 -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"
1000 loops, best of 5: 700 usec per loop
C:\code> pypy -m timeit -n 1000 -r 5 -c "[abs(x) for x in range(10000)]"
1000 loops, best of 5: 61.7 usec per loop
```

Otrzymane wyniki są zasadniczo takie same jak te dla wcześniejszych testów w tym rozdziale, przeprowadzanych na tych samych typach kodu. Przy badaniu wyrażenia `x ** 2` CPython 2.7 i PyPy są ponownie odpowiednio 10 razy i 100 razy szybsze niż CPython 3.3, co pokazuje, że wybór rodzaju timera nie jest tutaj istotnym czynnikiem. W przypadku funkcji `abs(x)`, którą mierzyliśmy wcześniej w timerze własnej roboty (`timeseqs.py`), te dwa Pythony są szybsze niż 3.3 odpowiednio o mały współczynnik i 10°, co sugeruje, że inna struktura kodu modułu `timeit` nie wpływa na porównania względne — rodzaj testowanego kodu w pełni określa rząd wielkości różnic prędkości.

Mała uwaga: zauważ, że wyniki ostatnich trzech testów, które są podobne do testów z użyciem własnego timera, są zasadniczo takie same jak wcześniej, ale wydają się pociągać za sobą niewielki narzut netto z powodu różnic w wykorzystaniu zasiegu — poprzednio korzystaliśmy z wcześniej zbudowanej listy, ale tutaj mamy albo generator w wersji 3.x, albo listę 2.x budowaną na nowo dla każdej wewnętrznej pętli podsumowującej. Innymi słowy, nie mierzmy dokładnie tego samego czasu, ale względne prędkości testowanych wersji Pythona są takie same.

Mierzenie czasu działania instrukcji wielowierszowych

Aby mierzyć czas działania wielowierszowych sekcji kodu w trybie wywołań *metod interfejsu API*, powinieneś użyć podziałów wierszy i tabulatorów lub spacji zgodnie z wymogami składni Pythona; kod odczytany z pliku źródłowego będzie to robić automatycznie. Ponieważ w tym trybie przekazujesz obiekty łańcuchowe Pythona do funkcji, nie musimy brać pod uwagę żadnych wymagań dotyczących powłoki, jednak w razie potrzeby powinieneś zachować ostrożność, aby uniknąć zag niezdanych cudzysłów. Poniżej zamieszczamy przykładowe czasy działania różnych rozwiązań pętli omawianych w rozdziale 13. (dla Pythona 3.3); możesz użyć tego samego wzorca do pomiaru czasu działania różnych sposobów odczytywania wierszy danych z plików, o których mówiliśmy w rozdziale 14.):

```
c:\code> py -3
>>> import timeit
>>> min(timeit.repeat(number=10000, repeat=3,
stmt="L = [1, 2, 3, 4, 5]\nfor i in range(len(L)): L[i] += 1"))
```

```

0.01397292797131814
>>> min(timeit.repeat(number=10000, repeat=3,
stmt="L = [1, 2, 3, 4, 5]\nfor i in range(len(L)):\n    L[i] += 1\n    ti += 1"))
0.015452276471516813
>>> min(timeit.repeat(number=10000, repeat=3,
stmt="L = [1, 2, 3, 4, 5]\nM = [x + 1 for x in L]"))
0.009464995838568635

```

Aby uruchamiać takie instrukcje wielowierszowe w trybie *wiersza poleceń*, powinieneś przekazywać każdy wiersz instrukcji jako osobny argument, z białymi spacjami reprezentującymi wcięcia — moduł `timeit` łączy wszystkie wiersze wraz ze znakami nowych wierszy między nimi, a później odtwarza wcięcia dla własnych potrzeb zagnieżdżania kodu. W tym trybie spacje wiodące mogą lepiej sprawdzać się przy tworzeniu wcięć niż tabulatory; powinieneś również pamiętać o odpowiednim cytowaniu wartości argumentów, jeżeli jest to wymagane przez powiokę:

```

c:\code> py -3 -m timeit -n 1000 -r 3 "L = [1,2,3,4,5]" "i=0" "while i < len(L):"
        "    L[i] += 1" "    i += 1"
1000 loops, best of 3: 1.54 usec per loop
c:\code> py -3 -m timeit -n 1000 -r 3 "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 0.959 usec per loop

```

Inne tryby użytkowania: instalacje, podsumowania i obiekty

Moduł `timeit` pozwala także na przygotowanie kodu *instalacyjnego*, który jest uruchamiany w zakresie głównego bloku kodu, ale którego czas wykonania nie jest naliczany do czasu wykonania testowanego bloku kodu — jest to potencjalnie przydatne do kodu inicjalizującego, który chcesz wykluczyć z całkowitego czasu działania, takiego jak importowanie wymaganych modułów, definiowanie testowanej funkcji i tworzenie zestawu danych testowych. Ponieważ są one uruchamiane w tym samym zakresie, wszelkie nazwy utworzone przez kod instalacyjny są dostępne dla głównego bloku testowanego kodu, podczas gdy nazwy zdefiniowane w sesji interaktywnej powłoki na ogół dostępne nie są.

Aby wskazać kod instalacyjny, w trybie wiersza poleceń użyj opcji `-s` (lub wielu takich opcji w przypadku konfiguracji wielowierszowych) i argumentu `setup` w trybie wywołania interfejsu API. Takie rozwiązanie pomoże Ci wyraźnie wskazać kod instalacyjny, którego działanie nie powinno być brane pod uwagę podczas mierzenia czasu działania testowanego bloku kodu. Prosta reguła mówi jednak, że im więcej kodu umieścisz w testowanym bloku kodu, tym bardziej otrzymane wyniki będą miarodajne i zbliżone do wydajności rzeczywistego kodu w zastosowaniach praktycznych:

```

c:\code> python -m timeit -n 1000 -r 3 "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 0.956 usec per loop
c:\code> python -m timeit -n 1000 -r 3 -s "L = [1,2,3,4,5]" "M = [x + 1 for x in L]"
1000 loops, best of 3: 0.775 usec per loop

```

Oto przykład wskazania kodu instalacyjnego w trybie wywołania interfejsu API: użyłem kodu przedstawionego poniżej do zmierzenia czasu wyznaczania wartości minimalnej z przykładu pokazanego w rozdziale 18. — uporządkowane zakresy są sortowane znacznie szybciej niż liczby losowe oraz szybciej niż wartości skanowane liniowo w kodzie przykładu dla wersji 3.3 (sąsiadujące łańcuchy znaków są tutaj łączone):

```

>>> from timeit import repeat
>>> min(repeat(number=1000, repeat=3,
setup='from mins import min1, min2, min3\n'
      'vals=list(range(1000))',
stmt= 'min3(*vals)'))
0.0387865921275079
>>> min(repeat(number=1000, repeat=3,
setup='from mins import min1, min2, min3\n'
      'import random\nvals=[random.random() for i in range(1000)]',
stmt= 'min3(*vals)'))
0.275656482278373

```

Za pomocą modułu `timeit` możesz również wyznaczyć całkowity czas działania, używać interfejsu API klasy modułu, mierzyć czas działania obiektów wywoływalnych zamiast ciągów instrukcji, akceptować automatyczne zliczanie pętli oraz używać technik opartych na klasach oraz dodatkowych przełączników wiersza poleceń i opcji argumentów interfejsu API, których nie będziemy tutaj omawiać — więcej szczegółowych informacji na te tematy znajdziesz w dokumentacji Pythona:

```
c:\code> py -3
>>> import timeit
>>> timeit.timeit(stmt='[x ** 2 for x in range(1000)]', number=1000)      # Czas całkowity
0.5238125259325834
>>> timeit.Timer(stmt='[x ** 2 for x in range(1000)]').timeit(1000)        # API klasy
0.5282652329644009
>>> timeit.repeat(stmt='[x ** 2 for x in range(1000)]', number=1000, repeat=3)
[0.5299034147194845, 0.5082454007998365, 0.5095136232504416]
>>> def testcase():
    y = [x ** 2 for x in range(1000)]                                     # Obiekty wywoływalne lub ciągi kodu
>>> min(timeit.repeat(stmt=testcase, number=1000, repeat=3))
0.5073828140463377
```

Moduł i skrypt testujący z użyciem modułu timeit

Zamiast zagłębiać się w więcej szczegółów na temat modułu `timeit`, przyjrzyjmy się programowi, który używa go do mierzenia czasu działania różnych rozwiązań, jak i różnych wersji Pythona. Poniższy plik, `pybench.py`, jest skonfigurowany do mierzenia czasu działania bloków instrukcji zakodowanych w skryptach, które go importują i używają w bieżącej wersji lub we wszystkich wersjach Pythona wymienionych na liście. Program korzysta z kilku narzędzi na poziomie aplikacji, o których opowiem już niebawem. Ponieważ dotyczy to głównie koncepcji, które już poznaliśmy i które są dobrze udokumentowane, pozostawimy to jako materiał do samodzielnej nauki oraz ćwiczenie z czytania kodu napisanego w języku Python.

```
"""
pybench.py: Testuje szybkość jednej lub więcej wersji Pythona na zestawie prostych testów
łańcuchów poleceń. Funkcja pozwala na zmianę badanych instrukcji. Moduł działa zarówno w wersji 2.x,
jak i 3.x, i może uruchamiać obie wersje.

Używa modułu timeit do testowania Pythona uruchamiającego ten skrypt za pomocą wywołań API
lub zestawu Pythonów poprzez odczytywanie danych z wiersza poleceń (os.popen) z flagą -m,
aby znaleźć odpowiedni plik w ścieżce wyszukiwania modułów.

Zamienia $listif3 na list () dla generatorów w wersji 3.x i pusty ciąg w wersji 2.x, więc 3.x działa
tak samo jak 2.x. Tylko w trybie wiersza poleceń należy podzielić instrukcje wielowierszowe na
oddzielne, cytowane

argumenty w wierszu, aby wszystkie mogły zostać połączone i uruchomione (w przeciwnym razie może
uruchamiać lub mierzyć czas działania tylko pierwszego wiersza), i zastąpić wszystkie tabulatory
wcięciem z 4 spacjami dla zachowania jednorodności.

Ostrzeżenia: tryb wiersza poleceń może nie działać poprawnie, jeżeli testowane instrukcje zawierają
podwójne cudzysłowy, cytowany ciąg instrukcji jest ogólnie niezgodny z powłoką lub wiersz poleceń
przekracza limit długości polecenia dla powłoki danej platformy – w takiej sytuacji użyj trybu
wywołania API lub własnego licznika czasu; program nie obsługuje jeszcze kodu instalacyjnego:
w obecnej wersji czas wszystkich instrukcji w testowanym bloku kodu jest wliczany do całkowitego
czasu działania.

"""

import sys, os, timeit
defnum, defrep= 1000, 5                                # Może się różnić w zależności od badanego kodu
def runner(stmts, pythons=None, tracecmd=False):
    """

    Główna część programu: uruchamia test zgodnie z parametrami wejściowymi;
    wywołanie odpowiada za tryb działania.
```

```

stmts: [(number?, repeat?, stmt-string)], zamienia $listif3 w polecenie
pythons: None=tylko bieżąca wersja, lub [(ispy3?, python-executable-path)]
"""

print(sys.version)
for (number, repeat, stmt) in stmts:
    number = number or defnum
    repeat = repeat or defrep           # 0=domyślnie
    if not pythons:
        # Uruchom stmt w tej wersji Pythona: wywołanie API
        # Nie ma potrzeby dzielenia wierszy kodu ani cytowania
        ispy3 = sys.version[0] == '3'
        stmt = stmt.replace('$listif3', 'list' if ispy3 else '')
        best = min(timeit.repeat(stmt=stmt, number=number, repeat=repeat))
        print('%.4f [%r]' % (best, stmt[:70]))
    else:
        # Uruchom stmt na wszystkich wersjach Pythona: wiersz poleceń
        # Dzielimy wiersze na szereg cytowanych argumentów
        print('-' * 80)
        print('[%r]' % stmt)
        for (ispy3, python) in pythons:
            stmt1 = stmt.replace('$listif3', 'list' if ispy3 else '')
            stmt1 = stmt1.replace('\t', ' ' * 4)
            lines = stmt1.split('\n')
            args = ' '.join(['"%s"' % line for line in lines])
            cmd = '%s -m timeit -n %s -r %s %s' % (python, number, repeat, args)
            print(python)
            if tracecmd: print(cmd)
            print('\t' + os.popen(cmd).read().rstrip())

```

Ten plik to tak naprawdę tylko połowa obrazu. Skrypty testujące używają funkcji tego modułu, przekazując konkretne, choć zmienne listy instrukcji i wersji Pythona do przetestowania, zgodnie z wymaganiami użytkownika. Na przykład poniższy skrypt, *pybench_cases.py*, testuje garść instrukcji i wersji Pythona oraz pozwala argumentom wiersza poleceń określić jego sposób działania: *-a* testuje wszystkie wymienione Pythony zamiast tylko jednego, a opcja *-t* śledzi utworzone wiersze poleceń, dzięki czemu można zobaczyć, w jaki sposób obsługiwane są instrukcje wielowierszowe i wcięcia (szczegółowe informacje znajdują się w notkach dokumentacyjnych obu plików):

```

"""
pybench_cases.py: Uruchamia pybench na zestawie Pythonów i instrukcji.

Wybierz tryby działania, edytując ten skrypt lub używając argumentów wiersza poleceń
(w sys.argv): np. uruchom polecenie
„C:\python27\python pybench_cases.py”, aby przetestować tylko jedną konkretną wersję na stmts,
„pybench_cases.py -a”, aby przetestować wszystkie wymienione Pythony lub
„py -3 pybench_cases.py -a -t”, aby śledzić również wiersze poleceń.
"""

import pybench, sys
pythons = [                                         # (ispy3?, ścieżka)
(1, 'C:\python33\python'),
(0, 'C:\python27\python'),

```

```

(0, 'C:\pypy\pypy-1.9\pypy')
]
stmts = [
    (0, 0, "[x ** 2 for x in range(1000)]"),           # (num,rpt,stmt)
    (0, 0, "res=[]\nfor x in range(1000): res.append(x ** 2)"),   # Iteracje
    (0, 0, "$listif3(map(lambda x: x ** 2, range(1000))))"),      # \n=wiele poleceń
    (0, 0, "list(x ** 2 for x in range(1000))"),          # \n\t=wcięcia
    (0, 0, "s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"), # $=lista lub ''
    (0, 0, "s = '?'\\nfor i in range(10000): s += '?'"),
]
tracecmd = '-t' in sys.argv                      # -t: śledzenie wiersza poleceń?
pythons = pythons if '-a' in sys.argv else None  # -a: wszystkie wersje z listy czy tylko
                                                # bieżąca?
pybench.runner(stmts, pythons, tracecmd)

```

Wyniki działania skryptu testującego

Oto wynik działania tego skryptu po uruchomieniu go w celu przetestowania określonej wersji (Python uruchamiający skrypt) — ten tryb wykorzystuje bezpośrednie wywołania API, a nie wiersz poleceń, z łącznym czasem podanym w lewej kolumnie, a testowaną instrukcją po prawej stronie. W pierwszych dwóch testach używamy programu uruchamiającego, aby zmierzyć czas działania wersji CPython 3.3 i 2.7, a w trzecim mierzmy wydajność dystrybucji PyPy w wersji 1.9:

```

c:\code> py -3 pybench_cases.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
0.5015 ['[x ** 2 for x in range(1000)]']
0.5655 ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.6044 ['list(map(lambda x: x ** 2, range(1000)))']
0.5425 ['list(x ** 2 for x in range(1000))']
0.8746 ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
2.8060 ["s = '?'\\nfor i in range(10000): s += '?'"]

c:\code> py -2 pybench_cases.py
2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)]
0.0696 ['[x ** 2 for x in range(1000)]']
0.1285 ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.1636 ['(map(lambda x: x ** 2, range(1000)))']
0.0952 ['list(x ** 2 for x in range(1000))']
0.6143 ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
2.0657 ["s = '?'\\nfor i in range(10000): s += '?'"]

c:\code> c:\pypy\pypy-1.9\pypy pybench_cases.py
2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit]
0.0059 ['[x ** 2 for x in range(1000)]']
0.0102 ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.0099 ['(map(lambda x: x ** 2, range(1000)))']
0.0156 ['list(x ** 2 for x in range(1000))']
0.1298 ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
5.5242 ["s = '?'\\nfor i in range(10000): s += '?'"]

```

Poniżej przedstawiamy wyniki działania tego skryptu po uruchomieniu w celu przetestowania *wielu wersji języka Python* dla każdego ciągu instrukcji. W tym trybie sam skrypt jest uruchamiany przez Pythona 3.3, ale uruchamia wiersze polecen powłoki, które uruchamiają inne wersje Pythona i moduł `timeit` w blokach testowanych instrukcji. W tym trybie musimy dzielić, formatować i cytować wyrażenia wielowierszowe do użycia w wierszach polecień zgodnie z oczekiwaniami czasowymi i wymaganiami powłoki.

Ten tryb opiera się również na opcji `-m` wiersza polecenia, pozwalającej Pythonowi na zlokalizowanie modułu `timeit` na ścieżce wyszukiwania modułów i uruchomienie go jako skryptu oraz użycie standardowych narzędzi bibliotecznych `os.popen` i `sys.argv` do odpowiednio uruchomienia polecenia powłoki i sprawdzenia argumentów wiersza polecenia. Więcej informacji na temat tych wywołań możesz znaleźć w dokumentacji Pythona i innych źródłach; metoda `os.popen` jest również krótko wspomniana podczas omawiania plików w rozdziale 9. i zastosowana w pętlach w rozdziale 13. Aby obserwować działanie linii poleczeń, powinieneś uruchomić skrypt z flagą `-t`:

```
c:\code> py -3 pybench_cases.py -a
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
-----
['[x ** 2 for x in range(1000)]']

C:\python33\python
    1000 loops, best of 5: 499 usec per loop

C:\python27\python
    1000 loops, best of 5: 71.4 usec per loop

C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 5.71 usec per loop
-----
['res=[]\nfor x in range(1000): res.append(x ** 2)']

C:\python33\python
    1000 loops, best of 5: 562 usec per loop

C:\python27\python
    1000 loops, best of 5: 130 usec per loop

C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 9.81 usec per loop
-----
 ['$listif3(map(lambda x: x ** 2, range(1000)))']

C:\python33\python
    1000 loops, best of 5: 599 usec per loop

C:\python27\python
    1000 loops, best of 5: 161 usec per loop

C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 9.45 usec per loop
-----
['list(x ** 2 for x in range(1000))']

C:\python33\python
    1000 loops, best of 5: 540 usec per loop

C:\python27\python
    1000 loops, best of 5: 92.3 usec per loop

C:\pypy\pypy-1.9\pypy
    1000 loops, best of 5: 15.1 usec per loop
-----
['s = \'spam\' * 2500\nx = [s[i] for i in range(10000)]']

C:\python33\python
```

```

        1000 loops, best of 5: 873 usec per loop
C:\python27\python
        1000 loops, best of 5: 614 usec per loop
C:\pypy\pypy-1.9\pypy
        1000 loops, best of 5: 118 usec per loop
-----
["s = '?'\\nfor i in range(10000): s += '?'"]
C:\python33\python
        1000 loops, best of 5: 2.81 msec per loop
C:\python27\python
        1000 loops, best of 5: 1.94 msec per loop
C:\pypy\pypy-1.9\pypy
        1000 loops, best of 5: 5.68 msec per loop

```

Jak widać, w większości tych testów CPython 2.7 jest wciąż szybszy niż CPython 3.3, a PyPy jest zauważalnie szybszy niż oba z nich — z wyjątkiem ostatniego testu, w którym PyPy jest dwa razy wolniejszy niż CPython, prawdopodobnie z powodu różnic w sposobie zarządzania pamięcią. Z drugiej strony wyniki czasowe bardzo często są w najlepszym razie względne; oprócz innych ogólnych zastrzeżeń dotyczących pomiarów czasu, o których mówiliśmy w tym rozdziale, powinieneś jeszcze uwzględnić następujące aspekty:

- Moduł `timeit` może zniekształcać wyniki w sposób wykraczający poza zakres zagadnień omawianych w tej książce (np. uwzględnienie wpływu działania mechanizmu automatycznego zwalniania nieużywanych zasobów).
- Zawsze istnieje pewien bazowy narzut czasowy, który różni się w zależności od wersji Pythona; jest tutaj ignorowany (ale w praktyce wydaje się być trywialnie mały).
- Skrypt uruchamia bardzo proste instrukcje, które mogą, ale nie muszą odzwierciedlać szybkości działania rzeczywistego kodu (ale nadal są poprawne).
- Wyniki mogą czasami różnić się w sposób, który wydaje się losowy (użycie czasu działania procesu może tutaj pomóc).
- Wszystkie przedstawione tutaj wyniki są bardzo podatne na zmiany związane z modyfikacjami Pythona (w rzeczywistości zmieniają się niemal w każdej nowej wersji Pythona!).

Innymi słowy, zawsze powinieneś wyciągnąć własne wnioski z tych liczb i uruchamiać testy na swoich Pythonach i komputerach, aby uzyskać wyniki bardziej odzwierciedlające działanie Twojego środowiska. Aby zmierzyć bazowe obciążenie każdego Pythona, uruchom `timeit` bez żadnych instrukcji lub równoważnie z instrukcją `pass`.

Jeszcze trochę zabawy z mierzeniem wydajności

Aby uzyskać więcej danych, spróbuj uruchomić skrypt na innych wersjach języka Python i innych ciągach testowych poleceń. W pliku `pybench_cases2.py` w pakiecie przykładów tej książki znajduje się jeszcze więcej testów pozwalających zobaczyć, jak CPython 3.3 wypada w porównaniu z wersją 3.2 czy jak PyPy 2.0 beta wypada w porównaniu z obecną wersją i jak wypadają inne przypadki użycia.

Wygrana funkcji `map` i rzadka porażka PyPy

Na przykład następujące testy w `pybench_cases2.py` mierzą wpływ obciążania innych operacji iteracyjnych wywołaniem funkcji, co zwiększa szansę funkcji `map` na wygraną, zgodnie z tym, o czym pisaliśmy we wcześniejszej wersji tego rozdziału — funkcja `map` zwykle traciła w takich testach poprzez domyślne skojarzenie z wywołaniami funkcji:

```

# pybench_cases2.py
pythons += [
    (1, 'C:\\python32\\python'),
    (0, 'C:\\pypy\\pypy-2.0-beta1\\pypy')]
stmts += [
    # Używamy wywołań funkcji: map wygrywa!
    (0, 0, "[ord(x) for x in 'spam' * 2500]"),
    (0, 0, "res=[]\\nfor x in 'spam' * 2500: res.append(ord(x))"),
    (0, 0, "$listif3(map(ord, 'spam' * 2500))"),
    (0, 0, "list(ord(x) for x in 'spam' * 2500)"),
    # Zbiory i słowniki

```

```

(0, 0, "{x ** 2 for x in range(1000)}"),
(0, 0, "s=set()\nfor x in range(1000): s.add(x ** 2)"),
(0, 0, "{x: x ** 2 for x in range(1000)}"),
(0, 0, "d={}\\nfor x in range(1000): d[x] = x ** 2"),
# Ciężki kaliber: liczba składająca się z ponad 300000 cyfr!
(1, 1, "len(str(2**1000000))")] # Tutaj PyPy przegrywa na całej linii

```

Oto wyniki działania tego skryptu w wersji CPython 3.x, pokazujące, w jaki sposób funkcja `map` staje się najszybsza, gdy w grę wchodzą wywołania funkcji wyrównujące jej szanse (funkcja `map` traciła wcześniej, gdy inne testy uruchamiały operację `x ** 2`):

```
c:\code> py -3 pybench_cases2.py
3.3.0 (v3.3.0:bd8af90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
0.7237 ["[ord(x) for x in 'spam' * 2500]"]
1.3471 ["res=[]\\nfor x in 'spam' * 2500: res.append(ord(x))"]
0.6160 ["list(map(ord, 'spam' * 2500))"]
1.1244 ["list(ord(x) for x in 'spam' * 2500)"]
0.5446 ['{x ** 2 for x in range(1000)}']
0.6053 ['s=set()\nfor x in range(1000): s.add(x ** 2)']
0.5278 ['{x: x ** 2 for x in range(1000)}']
0.5414 ['d={}\\nfor x in range(1000): d[x] = x ** 2']
1.8933 ['len(str(2**1000000))']
```

Tak jak poprzednio, w takich testach obecnie wersje 2.x działają szybciej niż 3.x, a dystrybucja PyPy jest jeszcze szybsza we wszystkich testach z wyjątkiem ostatniego, w którym jest gorsza o pełny rząd wielkości (10 razy), choć wszystkie pozostałe testy wygrywa w podobnym lub lepszym stosunku. Jeżeli uruchomisz testy plików wcześniej zakodowane w pliku `pybench_cases2.py`, przekonasz się, że PyPy traci także na korzyść CPythona podczas odczytywania plików wiersz po wierszu, tak jak w przypadku poniższej krotki testowej na liście `stmts`:

```
(0, 0, "f=open('C:/Python33/Lib/pdb.py')\\nfor line in f: x=line\\nf.close()"),
```

Ten test otwiera plik tekstowy o rozmiarze ok. 60 kB, składający się z 1675 wierszy, i odczytuje go wiersz po wierszu za pomocą iterаторów pliku. Petla wejściowa jest prawdopodobnie czynnikiem dominującym w całkowitym czasie testu. W tym teście CPython 2.7 jest dwa razy szybszy niż wersja 3.3, ale PyPy jest ogólnie o rząd wielkości wolniejszy niż dowolna wersja CPythona. Możesz znaleźć ten przypadek w plikach wyników `pybench_cases2` lub samodzielnie zweryfikować w sesji interaktywnej albo na poziomie wiersza poleceń (tak właśnie wewnętrznie działa `pybench`):

```
c:\code> py -3 -m timeit -n 1000 -r 5 "f=open('C:/Python33/Lib/pdb.py')"
"for line in f: x=line" "f.close()"
>>> import timeit
>>> min(timeit.repeat(number=1000, repeat=5,
stmt="f=open('C:/Python33/Lib/pdb.py')\\nfor line in f: x=line\\nf.close()"))
```

Inny przykład, który mierzy zarówno wydajność listy składanej, jak i bieżącą prędkość przetwarzania plików w dystrybucji PyPy, znajduje się w pliku `listcomp-speed.txt` w pakiecie przykładów z naszej książki; wykorzystuje bezpośrednie wiersze poleceń PyPy do uruchomienia kodu z rozdziału 14., co daje podobne wyniki: odczytywanie wierszy w PyPy jest dziś mniej więcej dziesięciokrotnie wolniejsze niż w dystrybucjach CPython.

Pominąłem tutaj wyniki działania innych wersji i dystrybucji Pythona zarówno ze względu na oszczędność miejsca, jak i dla tego, że takie wyniki mogą bardzo się zmienić do czasu, kiedy będziesz czytał te słowa. Jak zwykle różne typy kodu mogą wykazywać różną wydajność. Choć PyPy optymalizuje algorytmy wielu operacji, nie oznacza to wcale, że na pewno będzie w stanie zoptymalizować również Twój kod. Dodatkowe wyniki możesz znaleźć w pakiecie przykładów książki, ale zdecydowanie lepszym rozwiązaniem będzie samodzielne przeprowadzenie takich testów, co pozwoli Ci zweryfikować dzisiejsze ustalenia lub zaobserwować ich możliwe inne wyniki w przyszłości.

Jeszcze kilka słów o wpływie wywołań funkcji

Jak sugerowaliśmy wcześniej, funkcja `map` wygrywa również w kategorii używania *funkcji zdefiniowanych przez użytkownika* — następujące testy dowodzą prawdziwości naszego stwierdzenia, że funkcja `map` wygrywa taki wyścig w dystrybucjach CPython, jeżeli wywołania alternatywne muszą używać dowolnej innej funkcji:

```
stmts = [
(0, 0, "def f(x): return x\\n[f(x) for x in 'spam' * 2500]"),
```

```
(0, 0, "def f(x): return x\nres=[]\nfor x in 'spam' * 2500: res.append(f(x))"),
(0, 0, "def f(x): return x\n$listif3(map(f, 'spam' * 2500)))",
(0, 0, "def f(x): return x\nlist(f(x) for x in 'spam' * 2500)])]

c:\code> py -3 pybench_cases2.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
1.5400 ["def f(x): return x\n[f(x) for x in 'spam' * 2500]"]
2.0506 ["def f(x): return x\nres=[]\nfor x in 'spam' * 2500: res.append(f(x))"]
1.2489 ["def f(x): return x\nlist(map(f, 'spam' * 2500))"]
1.6526 ["def f(x): return x\nlist(f(x) for x in 'spam' * 2500)"]
```

Porównaj to z wynikami testów ord z poprzedniej sekcji; chociaż funkcje zdefiniowane przez użytkownika mogą być wolniejsze niż wbudowane, obecnie funkcje wydają się działać bardzo szybko, niezależnie od tego, czy są one wbudowane, czy nie. Zauważ, że całkowity czas tutaj obejmuje koszt wykonania funkcji pomocniczych, choć jest to tylko jedno wywołanie na każde 10 000 powtórzeń w pętli wewnętrznej — zatem na zdrowy rozsądek możemy przyjąć, że jest to czynnik zupełnie nieistotny dla przeprowadzanych testów.

Techniki porównywania — własne funkcje kontra moduł timeit

Dla nabrania lepszej perspektywy zobaczymy, jak wyniki oparte na module `timeit` w tej sekcji wypadają na tle wyników timera opartego na naszych własnych rozwiązaniach z poprzedniej sekcji; w tym celu uruchomimy plik `timeseqs3.py` z pakietu przykładów z tej książki, który używa naszego własnego rozwiązania pomiaru czasu, ale wykonuje tę samą operację $x^{**} 2$ i wykorzystuje te same liczby powtórzeń, co `pybench_cases.py`:

```
c:\code> py -3 timeseqs3.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
forLoop : 0.55022 => [0...998001]
listComp : 0.48787 => [0...998001]
mapCall : 0.59499 => [0...998001]
genExpr : 0.52773 => [0...998001]
genFunc : 0.52603 => [0...998001]

c:\code> py -3 pybench_cases.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)]
0.5015 ['[x ** 2 for x in range(1000)]']
0.5657 ['res=[]\nfor x in range(1000): res.append(x ** 2)']
0.6025 ['list(map(lambda x: x ** 2, range(1000)))']
0.5404 ['list(x ** 2 for x in range(1000))']
0.8711 ["s = 'spam' * 2500\nx = [s[i] for i in range(10000)]"]
2.8009 ["s = '?'\\nfor i in range(10000): s += '?'"]
```

Wyniki naszego własnego rozwiązania służącego do pomiaru czasu są bardzo podobne do wyników z tej sekcji opartych na skrypcie `pybench`, który wykorzystuje moduł `timeit`, choć nie są to w 100% identyczne testy — nasze własne pomiary oparte na skrypcie `timeseqs3.py` wykorzystują wywołanie funkcji w środkowej pętli i mają niewielki narzut czasowy na logikę samego timera, ale także korzystają w pętli wewnętrznej z wbudowanej listy zamiast z generatora `range` dostępnego w wersji 3.x, co wydaje się nieco przyspieszać działanie przy porównywanych testach (i nazwałybym ten przykład „testem zdraworozsądkowym”, ale nie jestem pewien, czy to określenie ma zastosowanie w jakiejkolwiek metodologii testów porównawczych!).

Możliwości ulepszenia — kod instalacyjny

Podobnie jak większość oprogramowania, przykłady programów przedstawionych w tej sekcji są otwarte i można je dowolnie rozszerzać. Na przykład pliki `pybench2.py` i `pybench2_cases.py` znajdujące się w pakiecie przykładów do książki dodają obsługę opisanej wcześniej opcji `setup` modułu `timeit` zarówno w trybie wywołań API, jak i wiersza poleceń.

Ta funkcja została początkowo pominięta ze względu na chęć zachowania zwięzości przykładów i, szczerze mówiąc, ponieważ moje testy jej nie wymagały, jej pominięcie dało bardziej kompletny obraz podczas porównywania szybkości działania różnych wersji Pythona, a w przypadku badania tylko jednej wersji narzut czasowy na działania konfiguracyjne jest dla każdego rozwiązania taki sam. Mimo to możliwość zdefiniowania kodu konfiguracyjnego, który jest uruchamiany tylko raz bez wliczania jego czasu działania do całkowitego czasu działania badanego kodu, jest bardzo przydatna — może to być na przykład import modułu, inicializacja obiektu lub definiowanie funkcji pomocniczych.

Nie będziemy tutaj zamieszczać tych dwóch plików w całości, a pokażemy tylko najważniejsze wprowadzone zmiany jako przykład praktycznej ewolucji oprogramowania — tak jak w przypadku testowanego polecenia, kod instalacyjny w trybie wywołania API jest przekazywany bez żadnych zmian, ale w trybie wiersza poleceń jest dzielony, uzupełniany odpowiednimi wcięciami i przekazywany z argumentem `-s` (`$listif3` nie jest używany, ponieważ czas działania kodu instalacyjnego nie jest uwzględniany w całkowitym czasie działania badanego polecenia):

```
# pybench2.py
...
def runner(stmts, pythons=None, tracecmd=False):
    for (number, repeat, setup, stmt) in stmts:
        if not pythons:
            ...
            best = min(timeit.repeat(
                setup=setup, stmt=stmt, number=number, repeat=repeat))
        else:
            setup = setup.replace('\t', ' ' * 4)
            setup = ' '.join(['-s "%s"' % line for line in setup.split('\n')])
            ...
            for (ispy3, python) in pythons:
                ...
                cmd = '%s -m timeit -n %s -r %s %s %s' %
                    (python, number, repeat, setup, args)
# pybench2_cases.py
import pybench2, sys
...
stmts = [                                     # (num,rpt,setup,stmt)
    (0, 0, "", "[x ** 2 for x in range(1000)]"),
    (0, 0, "", "res=[]\nfor x in range(1000): res.append(x ** 2)"),
    (0, 0, "def f(x):\n\treturn x",
     "[f(x) for x in 'spam' * 2500]"),
    (0, 0, "def f(x):\n\treturn x",
     "res=[]\nfor x in 'spam' * 2500:\n\tres.append(f(x))"),
    (0, 0, "L = [1, 2, 3, 4, 5]", "for i in range(len(L)): L[i] += 1"),
    (0, 0, "L = [1, 2, 3, 4, 5]", "i=0\nwhile i < len(L):\n\tL[i] += 1\n\ti += 1")
    ...
    pybench2.runner(stmts, pythons, tracecmd)
```

Uruchom ten skrypt z poziomu wiersza polecenia z flagami `-a i -t`, aby zobaczyć, jak budowane są wiersze wywołania dla kodu instalacyjnego. Na przykład następująca krotka specyfikacji testu generuje wiersz polecenia dla wersji 3.3 — być może utworzony kod nie jest zbyt oczywisty, ale wystarczający do przekazania polecenia z konsoli systemu Windows do modułu `timeit`, gdzie zostanie połączony ze znakami nowych wierszy, uzupełniony odpowiednimi wcięciami i wstawiony do wygenerowanej funkcji mierzącej czas działania:

```
(0, 0, "def f(x):\n\treturn x",
 "res=[]\nfor x in 'spam' * 2500:\n\tres.append(f(x))"
C:\python33\python -m timeit -n 1000 -r 5 -s "def f(x):" -s "      return x" "res=[]"
 "for x in 'spam' * 2500:" "      res.append(f(x))"
```

W trybie wywołań API ciągi kodu są przekazywane bez wprowadzania żadnych zmian, ponieważ nie ma potrzeby zaspokajania wymagań składni powłoki, zatem wystarczą normalne, osadzone w kodzie tabulatory i znaki końca wiersza. Spróbuj samodzielnie poeksperymentować, aby dowiedzieć się więcej o szybkości działania alternatywnych rozwiązań kodu w języku Python. Wcześniej czy później możesz w końcu napotkać jakieś ograniczenia powłoki dla większych sekcji kodu, zwłaszcza przy wywoływaniu w trybie wiersza poleceń, ale zarówno nasz własny timer, jak i tryb wywołań API oparty na module `timeit` i skrypcie `pybench` obsługują niemal dowolną ilość kodu. Testowanie szybkości działania kodu może być

świetną zabawą, ale kolejne ulepszenia musimy już pozostawić na przyszłość, jako sugerowane ćwiczenia do samodzielnego wykonania.

Inne zagadnienia związane z mierzeniem szybkości działania kodu — test `pystone`

W tym rozdziale koncentrujemy się na podstawowych zagadnieniach związanych z mierzeniem szybkości działania kodu, które mają zastosowanie ogólnie do testowania szybkości działania Pythona i które posłużyły jako baza przy opracowywaniu bardziej rozbudowanych przykładów z tej książki. Badanie szybkości działania Pythona jest jednak szerszą i bogatszą domeną, niż mogłoby się na pierwszy rzut oka wydawać. Jeżeli chcesz kontynuować ten temat, w internecie znajdziesz całe mnóstwo interesujących informacji. Wśród nich między innymi:

- `pystone.py` — program przeznaczony do pomiaru prędkości działania Pythona w całym zakresie kodu, dostarczany z Pythonem i zlokalizowany w katalogu `Lib\test`.
- <http://speed.python.org> — witryna projektu koordynującego prace nad popularnymi testami szybkości działania Pythona.
- <http://speed.pypy.org> — witryna testowa dystrybucji PyPy, spełniająca podobną rolę, co jej odpowiednik z poprzedniego punktu.

Na przykład test `pystone` jest oparty na programie testowym z języka C, który został przetłumaczony na Pythona przez samego twórcę Pythona, Guido van Rossumę. Zapewnia on inny sposób pomiaru względnych prędkości implementacji Pythona i ogólnie potwierdza nasze ustalenia z tego rozdziału:

```
c:\Python33\Lib\test> cd C:\python33\lib\test
c:\Python33\Lib\test> py -3 pystone.py
Pystone(1.1) time for 50000 passes = 0.685303
This machine benchmarks at 72960.4 pystones/second
c:\Python33\Lib\test> cd c:\python27\lib\test
c:\Python27\Lib\test> py -2 pystone.py
Pystone(1.1) time for 50000 passes = 0.463547
This machine benchmarks at 107864 pystones/second
c:\Python27\Lib\test> c:\pypy\pypy-1.9\pypy pystone.py
Pystone(1.1) time for 50000 passes = 0.099975
This machine benchmarks at 500125 pystones/second
```

Ponieważ nadszedł czas na podsumowanie tego rozdziału, powyższe zestawienie będzie w zupełności wystarczające jako niezależne potwierdzenie wyników naszych testów. Analizowanie znaczenia wyników testów `pystone` pozostawiamy jako sugerowane ćwiczenie do samodzielnego wykonania; kod testu nie jest identyczny w wersjach 3.x i 2.x, ale wydaje się, że obecnie różni się tylko pod względem operacji wyświetlania wyników i inicjalizacji zmiennych globalnych. Należy również pamiętać, że analiza porównawcza jest tylko jednym z wielu aspektów analizy kodu Pythona; więcej szczegółowych informacji na temat innych opcji w powiązanych domenach (np. testowania) znajdziesz w rozdziale 36., w sekcjach dotyczących przeglądu narzędzi programistycznych Pythona.

Pułapki związane z funkcjami

Zakończyliśmy rozdział poświęcony funkcjom, nadszedł więc czas na uświadomienie sobie kilku powszechnych pułapek związanych z tym zagadnieniem. Funkcje mają kilka zdradliwych cech, których mało kto się spodziewa. Wszystkie one są ukryte, a część z nich jest pomału usuwana z języka w najnowszych wydaniach, jednak stanowią poważny problem dla nowych użytkowników Pythona.

Lokalne nazwy są wykrywane w sposób statyczny

Jak wiemy, Python domyślnie klasyfikuje nazwy przypisywane w ciele funkcji jako *lokalne*, to znaczy zmiany w nich wprowadzane istnieją tylko w czasie działania funkcji. Jednak niewiele osób wie o tym, że Python wykrywa zmienne lokalne w sposób statyczny, na etapie komilacji definicji funkcji, nie w trakcie wykonania. Prowadzi to do szeregu dziwacznych zachowań, chętnie wytykanych przez początkujących programistów Pythona na listach dyskusyjnych.

Zazwyczaj nazwa, która nie została przypisana w funkcji, jest odczytywana z otaczającego ją modułu:

```
>>> X = 99
```

```

>>> def selector():      # Zmienna X jest użyta, ale nie jest przypisana
...     print(X)          # Zmienna X zostaje znaleziona w zasięgu globalnym...
>>> selector()
99

```

W powyższym przypadku nazwa X jest odczytywana z modułu. Zaobserwujmy jednak, co się stanie, jeżeli zastosujemy przypisanie do zmiennej X po jej odczytaniu w funkcji:

```

>>> def selector():
    print(X)            # Zmienna X jeszcze nie istnieje!
    X = 88              # Zmienna X zostaje zaklasyfikowana jako zmienna lokalna
                        # Analogiczna sytuacja może wystąpić dla "import X", "def X"
>>> selector()
UnboundLocalError: local variable 'X' referenced before assignment

```

Otrzymujemy błąd użycia zmiennej przed przypisaniem, ale przyczyna jest trudna do uchwycenia. Python, komplikując kod funkcji, napotyka przypisanie do nazwy X, więc decyduje, że jest to zmienna lokalna funkcji. Jednak na etapie wykonania funkcja print jest wywoływana przed przypisaniem, przez co zmienna lokalna X nie jest jeszcze zdefiniowana, co powoduje błąd. Zgodnie z regułami stosowania nazw w Pythonie, błąd informuje nas o tym, że zmienna lokalna została użyta, zanim została przypisana jej wartość. Każde przypisanie wartości do zmiennej w funkcji powoduje, że zmienna ta jest traktowana jako lokalna. Importy, przypisania, zagnieździone klasy i zagnieździone definicje funkcji są podatne na ten efekt uboczny.

Problem wynika z tego, że nazwy, do których przypisywane są wartości w ciele funkcji, są traktowane jako lokalne zmienne tej funkcji w całym jej zasięgu, nie tylko od miejsca przypisania. Przedstawiony przykład zawiera zatem niejasność: czy intencją programisty było wypisanie wartości zmiennej X zdefiniowanej globalnie w module, a następnie utworzenie zmiennej lokalnej X? A może rzeczywiście mamy do czynienia z błędem programisty? Python traktuje nazwę X jako lokalną w całym zasięgu funkcji, a taka sytuacja jest traktowana jako błąd. Jeżeli chcemy wypisać wartość zmiennej globalnej X, należy zadeklarować ją w funkcji jako globalną:

```

>>> def selector():
    global X           # Wymuszenie zmiennej X jako globalnej
    print(X)
    X = 88

>>> selector()
99

```

Należy jednak pamiętać, że oznacza to, iż przypisania do zmiennej X zmodyfikują wartość globalną, nie lokalną. W ramach funkcji nie możemy użyć globalnej zmiennej X, a następnie utworzyć lokalnej zmiennej o takiej samej nazwie. Aby móc użyć zmiennych w taki sposób, należy zastosować kwalifikację (odwołanie do nazwy w jej przestrzeni nazw), w tym przypadku importując w funkcji moduł, w którym jest ona zdefiniowana, i odwołując się do zmiennej globalnej X przez kwalifikację w ramach modułu:

```

>>> X = 99
>>> def selector():
    import __main__       # Zimportowanie własnego modułu
    print(__main__.x)     # Kwalifikowane odwołanie w celu uzyskania globalnej wersji nazwy
    X = 88               # Niekwalifikowane odwołanie użycie wersji lokalnej
    print(X)             # Wypisanie lokalnej wersji

>>> selector()
99
88

```

Kwalifikacja (odwołanie typu .x) pobiera wartość z przestrzeni nazw. W przypadku sesji interaktywnej przestrzeń nazw nosi nazwę __main__, zatem __main__.x odwołuje się do globalnej nazwy X. Szczegóły wyjaśniające tę zasadę można znaleźć w rozdziale 17.

W najnowszych wersjach Python został nieco naprawiony w tym zakresie i wypisuje trochę czytelniejsze komunikaty o błędach (dawniej wywoływał ogólny błąd nazwy), jednak sam mechanizm działa nadal w ten sam sposób (pułapka nadal istnieje).

Wartości domyślne i obiekty mutowalne

Jak wspominaliśmy pokrótko w rozdziale 17. i rozdziale 18., domyślne wartości argumentów mogą zachowywać stan między wywołaniami, choć często jest to nieoczekiwane. Zasadniczo domyślne wartości argumentów są określone i zapisywane tylko raz, gdy uruchamiana jest instrukcja `def`, a nie za każdym razem, gdy wywoływana jest funkcja wynikowa. Wewnętrznie Python zapisuje po jednym obiekcie na każdy domyślny argument dołączony do takiej funkcji.

Z reguły właśnie o to nam chodzi: ponieważ wartości domyślne są określone w czasie definiowania funkcji (instrukcja `def`), mogą w razie potrzeby otrzymywać wartości z otaczającego zasięgu (funkcje zdefiniowane w pętlach przez funkcje fabrykujące mogą nawet zależeć od tego zachowania — zobacz dalej). Jednak ponieważ wartość domyślna wykorzystuje ten sam obiekt przy każdym wywołaniu funkcji, należy zachować ostrożność w przypadku użycia zmiennych mutowalnych w charakterze wartości domyślnych. Na przykład poniższy kod wykorzystuje pustą listę jako wartość domyślną, a następnie zmienia tę wartość w miejscu przy każdym wywołaniu funkcji.

```
>>> def saver(x=[]):
    # Zapisuje pusty obiekt listy
    x.append(1)
    # Zmienia ten obiekt za każdym razem!
    print(x)

>>> saver([2])
# Wartość domyślna nie jest użyta
[2, 1]
>>> saver()
# Wartość domyślna jest użyta
[1]
>>> saver()
# Rośnie przy każdym wywołaniu!
[1, 1]
>>> saver()
[1, 1, 1]
```

Niektórzy programiści wykorzystują to zachowanie jako cechę języka — skoro mutowalne argumenty domyślne zachowują swój stan między wywołaniami, można ich używać w podobny sposób jak *staticznych* zmiennych lokalnych funkcji w języku C. W pewnym sensie mechanizm ten działa bardziej jak zmienne globalne, ale ich nazwy są lokalne w zasięgu funkcji, zatem nie kolidują z takimi samymi nazwami zdefiniowanymi gdzieś indziej.

Dla innych użytkowników ta cecha Pythona wydaje się być błędem, szczególnie gdy napotykają ją po raz pierwszy. Istnieją lepsze sposoby na zachowanie stanu między wywołaniami w Pythonie (np. za pomocą zagieźdzonych zasięgów domykających, które poznaliśmy w tej części, i klas, które będziemy omawiać w części VI).

Co więcej, mutowalne wartości domyślne są trudne do zapamiętania (i zrozumienia). Ich wartość jest zależna od kontekstu budowania obiektu funkcji. W poprzednim przykładzie w każdym wywołaniu funkcji wykorzystywany jest jeden obiekt listy — ten, który został utworzony na etapie wykonania instrukcji `def`. Oznacza to, że nie dostajemy nowej listy za każdym razem, gdy funkcja jest wywoływana, więc nie jest resetowana i rośnie z każdym wywołaniem metody `append`.

Jeżeli nie o takie zachowanie funkcji nam chodzi, wystarczy na początku funkcji wykonać kopię wartości domyślnej i pracować na tej kopii lub przenieść wyrażenie definiujące wartość domyślną do ciała funkcji. Dopóki wartość znajduje się w kodzie, który jest faktycznie wykonywany przy każdym uruchomieniu funkcji, za każdym razem otrzymamy zupełnie nowy obiekt:

```
>>> def saver(x=None):
    if x is None:
        # Nie został przekazany argument?
        x = []
    x.append(1)
    # Modyfikujemy nowy obiekt listy
    print(x)

>>> saver([2])
[2, 1]
>>> saver()
# Lista nie rośnie
[1]
>>> saver()
[1]
```

Nawiasem mówiąc, warunek `if` w powyższym przykładzie *niemal* mógłby zostać zastąpiony wyrażeniem `x = x or []`, które wykorzystuje fakt, że operator `or` zwraca wartość jednego ze swoich operandów: jeżeli żaden argument nie zostanie

przekazany, x otrzymuje domyślną wartość `None`, więc operator `or` zwróci pustą listę (drugi operand).

Jednak to nie będzie dokładnie to samo. Jeżeli w argumencie zostanie przekazana pusta lista, wyrażenie z operatorem `or` spowoduje, że funkcja zwróci nową listę, a nie tę, która została przekazana do funkcji, jak to się dzieje w przypadku użycia warunku `if` (wyrażenie zostaje rozwinięte do `x = [] or []`, co powoduje, że zwrócona zostaje wartość po prawej stronie operatora `or`; szczegółowo można znaleźć w sekcji „Testy prawdziwości i testy logiczne” w rozdziale 12.). Decyzja o zastosowaniu jednej z tych wersji powinna być uzależniona od rzeczywistych wymagań w stosunku do programu.

W Pythonie istnieje jeszcze inny sposób uzyskania efektu mutowalnych wartości domyślnych — w znacznie bardziej zrozumiałym sposobie, o którym mówiliśmy w rozdziale 19.

```
>>> def saver():
    saver.x.append(1)
    print(saver.x)

>>> saver.x = []
>>> saver()
[1]
>>> saver()
[1, 1]
>>> saver()
[1, 1, 1]
```

Nazwa funkcji jest nazwą globalną dla tej funkcji, ale nie musimy jej deklarować jako takiej, ponieważ nie modyfikujemy jej w danej funkcji. Mechanizm atrybutów działa zupełnie inaczej od domyślnych wartości argumentów, jednak ten sposób przywiązyania wartości do funkcji jest znacznie czytelniejszy dla programisty (i bez wątpienia mniej magiczny).

Funkcje, które nie zwracają wyników

W Pythonie funkcje nie muszą zwracać wyników: instrukcja `return` (lub `yield`) jest opcjonalna. Jeżeli funkcja nie zwraca wartości wprost za pomocą instrukcji `return`, działanie funkcji kończy się wraz z końcem kodu funkcji. Z technicznego punktu widzenia wszystkie funkcje zwracają wartość. Jeżeli nie zostanie ona określona w instrukcji `return`, wówczas funkcja automatycznie zwraca obiekt `None`.

```
>>> def proc(x):
    print(x)                      # Brak instrukcji return powoduje zwrócenie wartości None

>>> x = proc('testing 123...')
testing 123...
>>> print(x)
None
```

Funkcje, które nie używają polecenia `return`, takie jak powyższa, są w Pythonie odpowiednikiem „procedur” znanych z innych języków programowania. Procedury są zazwyczaj wywoływanie jak instrukcje, a zwracany obiekt `None` jest ignorowany, ponieważ wykonują swoje działania bez zwracania jakiegoś użytecznego wyniku.

Warto zdawać sobie sprawę z tej specyfiki języka, ponieważ Python nie informuje o tym, że próbujemy użyć wyniku funkcji, która nie zwraca żadnej wartości. Na przykład, jak wspominaliśmy w rozdziale 11., przypisanie wyniku działania metody `append` listy nie wywoła błędu, ale z powrotem otrzymamy obiekt `None`, a nie (jak można by się spodziewać) zmodyfikowaną listę:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)        # append jest "procedurą"
>>> print(list)                # append modyfikuje listy w miejscu
None
```

Jak wspomniałem w punkcie „Często spotykane problemy programistyczne” w rozdziale 15., funkcje niezwracające wyniku wykonują swoje zadania jako efekt uboczny i powinny być stosowane w charakterze instrukcji, a nie wyrażenia.

Różne problemy związane z funkcjami

Oto dwa dodatkowe błędy związane z funkcjami, które są wystarczająco powszechnne, aby je tutaj przytoczyć.

Otaczanie zasięgów i zmiennych pętli: funkcje fabrykujące

Tę pułapkę opisywaliśmy w rozdziale 17. w dyskusji na temat otaczania zasięgów funkcji, ale dla przypomnienia: tworząc funkcje fabrykujące (zwane również domknięciami), należy zachować ostrożność z zasięgami funkcji otaczającymi zmienne, które mogą być modyfikowane przez otaczające pętle — kiedy generowane funkcje są później wywoływane, wszystkie takie referencje przechowują wartość *ostatniej* iteracji pętli w zasięgu funkcji fabrykującej. W takim przypadku należy użyć wartości domyślnych do zapisania wartości zmiennych pętli, zamiast polegać na automatycznym wyszukiwaniu w otaczających zasięgach. Zobacz sekcję „Zmienne pętli mogą wymagać wartości domyślnych, a nie zasięgów” w rozdziale 17., gdzie znajdziesz więcej szczegółowych informacji na ten temat.

Ukrywanie wbudowanych funkcji przez przypisania: cieniowanie

W rozdziale 17. pokazywaliśmy również, jak można ponownie przypisać wbudowane nazwy w zasięgu lokalnym lub globalnym; takie ponowne przypisanie skutecznie ukrywa i zastępuje wbudowaną nazwę na pozostałą część zasięgu, w którym następuje przypisanie. Oznacza to, że nie będzie można użyć oryginalnej wbudowanej wartości nazwy. Dopóki nie potrzebujesz wbudowanej, oryginalnej wartości przypisywanej nazwy, nie stanowi to problemu — istnieje wiele nazw wbudowanych i można je dowolnie wykorzystywać ponownie. Jeżeli jednak przypadkowo zmienisz przypisanie wbudowanej nazwy, na której opiera się Twój kod, możesz mieć problemy. Nie powinieneś więc tego robić; albo przynajmniej staraj się używać narzędzi takich jak *PyChecker*, które mogą Cię ostrzec, jeżeli to zrobisz. Dobrą wiadomością jest to, że często używając wbudowanych nazw Pythona, szybko się ich nauczysz, używanie ich stanie się Twoją drugą naturą, a mechanizmy wyłapywania błędów w Pythonie będą Cię ostrzegać już na wczesnych etapach testowania, kiedy użyte przez Ciebie nazwy nie do końca będą tym, o czym myślałeś.

Podsumowanie rozdziału

Ten rozdział był dopełnieniem naszego przeglądu funkcji i wbudowanych narzędzi iteracyjnych dzięki większemu studiu przypadku, gdzie mierzyliśmy wydajność alternatywnych sposobów implementacji iteracji oraz sprawdzaliśmy szybkość działania różnych wersji Pythona. Rozdział zakończył się przeglądem typowych błędów związanych z funkcjami, dzięki czemu będziesz mógł uniknąć takich pułapek w przyszłości. Nasza opowieść o iteracjach będzie po raz ostatni kontynuowana w części VI, gdzie nauczysz się kodować zdefiniowane przez użytkownika obiekty iterowalne, które generują wartości za pomocą klas i metody `__iter__` (rozdział 30., w sekcjach dotyczących przeciążenia operatora).

W tym miejscu kończy się część książki poświęcona narzędziom funkcyjnym. W następnej części rozszerzymy naszą wiedzę na temat *modułów*, które są plikami narzędzi tworzącymi najwyższą jednostkę organizacyjną programów w Pythonie, a także strukturą, w której zawsze działają tworzone przez nas funkcje. Następnie zajmiemy się klasami, czyli narzędziami będącymi swego rodzaju pakietami funkcji ze specjalnymi pierwszymi argumentami. Jak zobaczymy, klasy definiowane przez użytkownika mogą implementować obiekty realizujące protokół iteracji, podobnie jak omawiane w tym rozdziale generatorы i inne obiekty iterowalne. W rzeczywistości wszystko, czego nauczyłeś się w tej części książki, będzie miało zastosowanie, gdy funkcje będą się później pojawiały w kontekście metod klasowych.

Jednak zanim przejdziemy do omawiania modułów, powinieneś poświęcić chwilę czasu na rozwiązywanie quizu z tego rozdziału i wykonanie ćwiczeń dla tej części książki, aby sprawdzić to, czego się nauczyłeś o funkcjach.

Sprawdź swoją wiedzę — quiz

1. Jaki wniosek można wyciągnąć z tego rozdziału na temat względnej prędkości działania narzędzi iteracyjnych Pythona?
2. Jaki wniosek można wyciągnąć z tego rozdziału na temat względnej prędkości działania różnych wersji samego Pythona?

Sprawdź swoją wiedzę — odpowiedzi

1. Zasadniczo listy składane są zwykle najszybsze w całej grupie; funkcja `map` wygrywa z nimi w Pythonie tylko wtedy, gdy wszystkie narzędzia iteracyjne muszą wywoływać jakieś funkcje; pętle `for` są zwykle wolniejsze niż wyrażenia składane; funkcje i wyrażenia generatora są wolniejsze niż złożenia o stały współczynnik. W dystrybucji PyPy niektóre z tych wyników różnią się; na przykład funkcja `map` ma często inną wydajność względową, a listy składane wydają się być zawsze najszybsze, co może być spowodowane lepszą optymalizacją na poziomie funkcji.

Przynajmniej tak jest obecnie w testowanych wersjach Pythona, na danej maszynie testowej i dla kodu mierzącego czas działania — wyniki mogą się różnić, jeżeli którykolwiek z tych trzech czynników się zmieni. W celu uzyskania najbardziej miarodajnych wyników powinieneś użyć własnego modułu `timer` lub standardowej

biblioteki `timeit`. Pamiętaj również, że iteracje są tylko jednym z elementów mających wpływ na ogólny czas działania programu, stąd testowanie większej ilości kodu daje pełniejszy obraz.

2. Ogólnie dystrybucja PyPy 1.9 (implementująca Pythona 2.7) jest zazwyczaj szybsza niż CPython 2.7, a CPython 2.7 jest często szybszy niż CPython 3.3. W większości przypadków PyPy jest około 10 razy szybszy niż CPython, a CPython 2.7 jest zwykle nieco tylko szybszy niż CPython 3.3. W przypadkach, w których korzystamy z matematyki liczb całkowitych, CPython 2.7 może być 10 razy szybszy niż CPython 3.3, a PyPy może być nawet 100 razy szybszy niż wersja 3.3. W innych przypadkach (np. operacje na łańcuchach znaków czy korzystanie z iteratorów plików) PyPy może być nawet dziesięciokrotnie wolniejszy niż CPython, chociaż moduł `timeit` sposob zarządzania pamięcią mogą znacząco wpływać na niektóre wyniki. Test porównawczy `pystone` potwierdza nasze względne rankingi, chociaż rozmiar raportowanych różnic nie są zgodne ze względem sposob kodowania.

Przynajmniej tak jest obecnie w testowanych wersjach Pythona, na danej maszynie testowej i dla kodu mierzącego czas działania — wyniki mogą się różnić, jeżeli którykolwiek z tych trzech czynników się zmieni. Dla uzyskania najbardziej miarodajnych wyników powinieneś użyć własnego modułu `timer` lub standardowej biblioteki `timeit`. Jest to szczególnie prawdziwe w przypadku testowania szybkości działania samego Pythona, która może być optymalizowana w każdej nowej wersji.

Sprawdź swoją wiedzę – ćwiczenia do części czwartej

W poniższych ćwiczeniach zaczniesz już pisać nieco bardziej wyrafinowane programy. Swoje rozwiązania powinieneś skonsultować z odpowiedziami w sekcji „Część IV. Funkcje i generatory” w dodatku D. Kod programów powinieneś zapisywać w plikach modułów, dzięki czemu w przypadku popełnienia błędu w sesji interaktywnej nie będziesz musiał przepisywać wszystkiego od początku.

1. *Podstawy.* W sesji interaktywnej Pythona napisz funkcję przyjmującą pojedynczy argument i wyświetlającą na ekranie jego wartość, a następnie wywołaj ją ręcznie, przekazując argumenty różnych typów: ciąg znaków, liczbę całkowitą, listę, słownik. Następnie wywołaj tę funkcję, nie podając żadnego argumentu. Co się wówczas stanie? A co się stanie, kiedy wywołując tę funkcję, podasz dwa argumenty?
2. *Argumenty.* Napisz funkcję `adder` i zapisz ją w pliku modułu. Funkcja powinna przyjmować dwa argumenty i zwracać ich sumę (lub połączenie). Następnie na końcu modułu dopisz kod testujący, wywołujący tę funkcję z argumentami różnych typów (dwa ciągi znaków, dwie listy, dwie liczby zmiennoprzecinkowe) i wywołaj moduł z wiersza poleceń jako skrypt. Czy istnieje konieczność wywoływania funkcji `print` w celu wyświetlenia na ekranie wyników tych wywołań?
3. *Zmienna liczba argumentów.* Spróbuj uogólnić funkcję `adder` utworzoną w poprzednim punkcie, tak aby obliczała sumę dowolnej liczby argumentów, i zmień wywołania testowe w taki sposób, aby wykorzystywały więcej niż jeden argument. Jakiego typu jest zwracana suma? (Wskazówka: wycinek `S[:0]` zwraca pustą sekwencję tego samego typu co `S`, a funkcja wbudowana `type` może służyć do sprawdzania typów. Prostsze rozwiązanie możesz znaleźć w rozdziale 18. w przykładzie własnej implementacji funkcji `min`). Co się stanie, gdy zostaną przekazane argumenty różnych typów? A co, jeżeli przekażemy słowniki?
4. *Słowa kluczowe.* Zmodyfikuj funkcję `adder` napisaną w punkcie 2. w taki sposób, aby akceptowała trzy argumenty i wykonywała na nich operację sumowania lub łączenia: `def adder(good, bad, ugly)`. Następnie zdefiniuj wartości domyślne każdego z tych argumentów i poeksperymentuj z tą funkcją w sesji interaktywnej. Spróbuj wywołać tę funkcję z jednym, dwoma, trzema i czterema argumentami. Następnie przekaż argumenty ze słowami kluczowymi. Czy zadziała wywołanie `adder(ugly=1, good=2)`? Dlaczego tak się dzieje? Uogólnij funkcję `adder` w taki sposób, aby akceptowała dowolną liczbę argumentów ze słowami kluczowymi. Zadanie jest podobne do punktu 3., z tą różnicą, że powinieneś iterować po słowniku, nie po krotce argumentów. (Wskazówka: metoda `dict.keys` zwraca listę, po której można przemieszczać się w pętli `for` lub `while`, ale w Pythonie 3.x wynik tej metody należy przekształcić na listę, zanim odwołamy się do jej elementów po indeksie; metoda `dict.values` również może Ci tutaj pomóc).
5. *Narzędzia słowników.* Napisz funkcję o nazwie `copyDict(dict)` kopującą argument w postaci słownika. Funkcja powinna zwrócić nowy słownik zawierający wszystkie elementy swojego argumentu. Użyj metody `keys` słownika do przemieszczania się po kluczach słownika (od Pythona 2.2 można iterować po kluczach słownika bez wywołania metody `keys`). Kopiowanie sekwencji jest proste (`X[:]` wykonuje kopię najwyższego poziomu). Czy taka metoda działa również dla słowników? Jak wyjaśniono w rozwiązaniu tego ćwiczenia, ponieważ słowniki są już obecnie wyposażone w podobne narzędzia, więc to i następne ćwiczenie są tylko ćwiczeniami w kodowaniu, choć nadal służą jako reprezentatywne przykłady funkcji.
6. *Narzędzia słowników.* Napisz funkcję `addDict(dict1, dict2)` generującą złączenie dwóch słowników. Funkcja powinna zwracać nowy słownik zawierający wszystkie wartości z obydwu argumentów (przyjmujemy, że będą to słowniki). Jeżeli w każdym z argumentów występuje ten sam klucz, w wyniku może znaleźć się dowolna z wartości. Przetestuj funkcję za pomocą kodu testującego w zawierającym ją module, kod powinien być uruchamiany w przypadku wywołania funkcji w formie skryptu. Co się stanie, jeżeli zamiast słowników zostaną przekazane listy? W jaki sposób można uogólnić funkcję, aby obsłużyła również ten przypadek? (Wskazówka: przydatna może okazać się funkcja `type` wspomniana wcześniej. Czy kolejność przekazanych argumentów ma znaczenie?).

7. Więcej przykładów dopasowywania argumentów. Najpierw zdefiniuj sześć funkcji przedstawionych poniżej (możesz to zrobić w sesji interaktywnej lub w pliku modułu, który będziesz mógł następnie zimportować):

```
def f1(a, b): print(a, b)          # Normalne argumenty
def f2(a, *b): print(a, b)          # Zmienna liczba argumentów pozycyjnych
def f3(a, **b): print(a, b)          # Zmienna liczba słów kluczowych
def f4(a, *b, **c): print(a, b, c)    # Tryby mieszane
def f5(a, b=2, c=3): print(a, b, c)    # Wartości domyślne
def f6(a, b=2, *c): print(a, b, c)    # Zmienna liczba argumentów pozycyjnych i wartości
domyślnych
```

Przetestuj te funkcje w sesji interaktywnej i wyjaśnij otrzymane wyniki. W niektórych przypadkach przydatny może okazać się algorytm wydobywania argumentów funkcji opisany w rozdziale 18. Czy mieszanie trybów dopasowania argumentów jest dobrym pomysłem? Czy można sobie wyobrazić przypadki, gdy taka technika będzie użyteczna?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)
>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)
>>> f5(1)
>>> f5(1, 4)
>>> f6(1)
>>> f6(1, 3, 4)
```

8. Powrót do liczb pierwszych. Poniższy fragment kodu jest powtórką z rozdziału 13. Jest to uproszczony algorytm weryfikujący, czy podana liczba jest liczbą pierwszą:

```
x = y // 2                                # Dla y > 1
while x > 1:
    if y % x == 0:                         # Reszta z dzielenia
        print(y, 'dzieli się przez', x)
        break                               # Pomijamy resztę
    x -= 1
else:                                         # Normalne wyjście
    print(y, 'jest liczbą pierwszą')
```

Zapisz ten kod jako funkcję w module do ponownego użycia (y powinien być przekazywanym argumentem), a w module umieść kod testujący w przypadku uruchomienia modułu jako skryptu. Poeksperymentuj, zastępując operator dzielenia całkowitego `//` z pierwszego wiersza kodu, aby przekonać się, w jaki sposób w wersji 3.x zmieniła się semantyka zwykłego dzielenia `(/)`, co może powodować efekty uboczne w kodzie napisanym dla poprzednich wersji Pythona (dalejsze informacje na ten temat można znaleźć w rozdziale 5.). Co można zrobić z liczbami ujemnymi oraz z 0 i 1? W jaki sposób przyspieszyć ten kod? Wyniki wywołania funkcji powinny wyglądać następująco:

```
13 jest liczbą pierwszą
13.0 jest liczbą pierwszą
15 dzieli się przez 5
15.0 dzieli się przez 5.0
```

9. Iteracje i listy składane. Napisz kod budujący nową listę zawierającą pierwiastki kwadratowe wszystkich liczb z listy [2, 4, 9, 16, 25]. Zakoduj to najpierw jako pętlę `for`, następnie jako wywołanie funkcji `map`, następnie jako listę składaną, a na koniec jako wyrażenie generatora. Wykorzystaj funkcję `sqrt` ze standardowego modułu `math` (zainportuj moduł `math` i użyj wywołania `math.sqrt(X)`). Które z tych podejść lubisz najbardziej?

10. Narzędzia do pomiaru czasu. W rozdziale 5. pokazywaliśmy trzy sposoby obliczania pierwiastków kwadratowych: `math.sqrt(X)`, `X ** .5` oraz `pow(X, .5)`. Jeżeli w pisany programie jesteś zmuszony do wykonywania dużej ilości tego typu obliczeń, różnice w wydajności poszczególnych rozwiązań mogą mieć znaczący wpływ na wydajność całego programu. Aby przekonać się, która wersja jest najszybsza, zmodyfikuj skrypt `timerseqs.py`, który napisaliśmy w tym rozdziale, tak aby mierzył czas działania każdego z tych trzech

narzędzi. Do testowania używaj funkcji `bestof` lub `bestoftotal` z jednego z modułów `timer` z tego rozdziału (możesz użyć wersji oryginalnej, wariantu zawierającego tylko słowa kluczowe `3.x` lub wersji `2.x/3.x`, albo po prostu użyć modułu `timeit` Pythona). Możesz również rozbudować kod testujący w tym skrypcie, tak aby uzyskać lepsze możliwości ponownego użycia — na przykład poprzez przekazanie krótkiej funkcji testowej do ogólnej funkcji testującej (w tym ćwiczeniu możesz użyć podejścia typu `kopij-wklej-zmodyfikuj`). Który sposób obliczania wartości pierwiastka kwadratowego wydaje się działać najszybciej na Twoim komputerze i używanej wersji Pythona? Jak możesz w interaktywny sposób zmierzyć szybkość działania słowników składanych i porównać ją z szybkością działania pętli `for`?

11. *Funkcje rekurencyjne.* Napisz prostą funkcję rekurencyjną o nazwie `countdown`, która wyświetla kolejne liczby, odliczając do zera. Na przykład wywołanie funkcji `countdown(5)` powinno wyświetlić: 5 4 3 2 1 stop. Nie ma oczywistych powodów, dla których miałbyś kodować taką funkcję z użyciem jawnego stosu lub kolejki, ale co z podejściem niefunkcyjnym? Czy generator miałby tutaj sens?
12. *Obliczanie silni.* Wreszcie klasyk informatyki (ale mimo to bardzo ilustracyjny). Pojęcie silni zastosowaliśmy podczas omawiania permutacji w rozdziale 20.: $N!$, obliczane jako $N * (N-1) * (N-2) * \dots * 1$. Na przykład $6! = 6 * 5 * 4 * 3 * 2 * 1 = 720$. Napisz cztery opisane dalej funkcje obliczające wartość silni i zmierz ich szybkość działania: (1) funkcja rekurencyjna, taka jak opisywaliśmy w rozdziale 19.; (2) funkcja korzystająca z wywołania funkcji `reduce`, tak jak opisywaliśmy w rozdziale 19.; (3) funkcja z prostą iteracyjną pętlą licznika zgodnie z rozdziałem 13.; (4) funkcja wykorzystująca narzędzie biblioteczne `math.factorial`, opisywane w rozdziale 20. Użyj modułu `timeit`, opisywanego w rozdziale 21., do sprawdzenia szybkości działania każdej z funkcji. Jakie wnioski możesz wyciągnąć na podstawie wyników?

[1] Zwróć uwagę, jak musimy tutaj ręcznie przekazywać funkcje do timera. W rozdziale 39. i rozdziale 40. zobaczyłeś alternatywne timery oparte na *dekoratorach*, z którymi funkcje czasowe są wywoływane w normalny sposób, ale wymagają dodatkowej składni @. Dekoratory mogą być bardziej przydatne do funkcji narzędziowych z logiką czasową, gdy są używane w większym systemie i nie obsługują tak łatwo przyjętych tutaj bardziej izolowanych wzorców wywołań testowych — po dekorowaniu *każde* wywołanie funkcji uruchamia logikę czasową, co może być wadą lub zaletą w zależności od Twoich celów.

Część V Moduły i pakiety

Rozdział 22. Moduły — wprowadzenie

Niniejszy rozdział rozpoczyna nasze pogłębione omówienie *modułu* w Pythonie — jednostki najwyższego poziomu organizacji programu, która pakuje razem kod programu i dane w celu późniejszego, ponownego ich użycia oraz zapewnia niezależne przestrzenie nazw, które minimalizują konflikty nazw zmiennych w Twoich programach. Moduły zazwyczaj odpowiadają plikom programów Pythona. Każdy plik jest modułem, a moduły importują inne moduły w celu skorzystania z zawartych w nich nazw. Modułami mogą być również rozszerzenia napisane w językach zewnętrznych, takich jak C, Java czy C#, a nawet katalogi w poleceniach importu w pakietach. Moduły przetwarzane są za pomocą dwóch instrukcji oraz jednej ważnej funkcji:

`import`

Pozwala klientowi (plikowi importującemu) pobrać moduł jako całość.

`from`

Pozwala klientom pobierać określone nazwy z modułu.

`imp.reload` (w wersji 2.x `reload`)

Umożliwia przeładowanie kodu modułu bez zatrzymywania Pythona.

W rozdziale 3. przedstawiono podstawy modułów i od tego czasu regularnie z modułów korzystamy. Naszym celem jest rozszerzenie podstawowych koncepcji modułów, które już znasz, a następnie przejście do ich bardziej zaawansowanych zastosowań. Ten pierwszy rozdział zawiera omówienie podstawowych zagadnień związanych z modułami i ogólne spojrzenie na rolę modułów w całej strukturze programu. W kolejnym rozdziale oraz kilku następnych zagłębimy się w szczególne programistyczne stojące za teorią.

Po drodze zapoznamy się ze szczegółami dotyczącymi modułów, o których do tej pory jeszcze nie mówiliśmy — dowiesz się tutaj między innymi o przeładowywaniu modułów, atrybutach `_name_` oraz `_all_`, importowaniu pakietów, składni importowania wzelnego, przestrzeni nazw pakietów w wersji 3.3 i wielu innych. Ponieważ moduły oraz klasy są jedynie wyróżnionymi *przestrzeniami nazw* (ang. *namespaces*), powrócimy również do koncepcji związanych z samymi przestrzeniami nazw.

Dlaczego używamy modułów

W skrócie, moduły udostępniają łatwy sposób organizowania komponentów w systemy, służąc jako samodzielne pakiety zmiennych znane jako *przestrzenie nazw*. Wszystkie zmienne zdefiniowane na najwyższym poziomie pliku modułu stają się atrybutami zaimportowanego obiektu modułu. Jak widzieliśmy w poprzedniej części książki, importowanie daje dostęp do nazw z zakresu globalnego modułu. Oznacza to, że zakres globalny pliku modułu *staje się* po zaimportowaniu przestrzeń nazw atrybutów obiektu modułu. Moduły Pythona pozwalają na łączenie pojedynczych plików w większe systemy programów.

Co więcej, z ogólnego punktu widzenia moduły pełnią przynajmniej trzy role.

Ponowne wykorzystanie kodu

Jak wspomniano w rozdziale 3., moduły pozwalają na stałe zachować kod w plikach. W przeciwieństwie do kodu wpisanego w interaktywnym wierszu poleceń Pythona, który znika w momencie zakończenia Pythona, kod z modułów pozostaje i jest *trwały* — można go przeładować i wykonać ponownie tyle razy, ile potrzebujemy. Co ważniejsze, moduły są miejscem definiowania nazw, znanych jako *atrybuty*, do których może się odwoływać wiele klientów zewnętrznych. Przy dobrym użyciu wspierają również (nomen omen) *modułową* konstrukcję programu, która grupuje funkcjonalność w jednostki wielokrotnego użytku.

Dzielenie przestrzeni nazw systemu

Moduły są również w Pythonie jednostką organizacyjną najwyższego poziomu. Chociaż zasadniczo są to tylko pakiety nazw, to łączą one zmienne w *samodzielne* pakiety — dopóki w jawnym sposobie nie zaimportujemy pliku, nie zobaczymy jego zmiennych. Podobnie jak w przypadku lokalnych zasięgów funkcji, takie rozwiązywanie pomaga uniknąć konfliktów nazw w programach. W praktyce nie możemy się tego ustrzec — tak naprawdę wszystko „żyje” w module i zarówno uruchamiany kod, jak i tworzone obiekty są zawsze w niewiadomy sposób zamknięte w module. Z tego powodu moduły są naturalnymi narzędziami do grupowania komponentów systemu.

Implementowanie współdzielonych usług oraz danych

Z operacyjnego punktu widzenia moduły przydają się również do implementowania komponentów, które są współdzielone w całym systemie i tym samym wymagają tylko *jednej kopii*. Jeżeli na przykład chcemy udostępnić obiekt globalny, który będzie wykorzystywany przez więcej niż jedną funkcję lub plik, możemy umieścić go w module, który następnie będzie mógł zostać zaimportowany przez wiele klientów.

Tak przynajmniej wygląda nieco abstrakcyjna historia modułów — aby jednak naprawdę zrozumieć ich rolę w Pythonie, musimy poczynić pewną dygresję i omówić ogólną strukturę programu w języku Python.

Architektura programu w Pythonie

Dotychczas w opisach programów Pythona w książce starałem się nieco łagodzić ich rzeczywisty stopień skomplikowania. W praktyce programy zazwyczaj obejmują więcej niż jeden plik. Większość programów, poza może najprostszymi skryptami, przybiera formę systemów składających się z *wielu plików* — tak jak mogłeś się o tym przekonać w przypadku programów do pomiaru czasu, o których mówiliśmy w poprzednim rozdziale. Nawet jeżeli udałoby Ci się zmieścić swój program w jednym pliku, z pewnością kiedyś będziesz pisać programy korzystające z bibliotek i modułów zewnętrznych napisanych przez inne osoby.

W niniejszym podrozdziale omówimy ogólną *architekturę* programów napisanych w języku Python — sposób dzielenia programu na zbiór plików źródłowych (inaczej modułów) oraz łączenie poszczególnych części w całość. Jak będziesz mógł się przekonać, Python wspiera modułową strukturę programu, która grupuje funkcjonalność w spójne i nadające się do wielokrotnego użytku jednostki w zupełnie naturalny i niemal automatyczny sposób. Po drodze poznasz również najważniejsze zagadnienia dotyczące modułów Pythona, importowania oraz atrybutów obiektów.

Struktura programu

Na poziomie podstawowym program napisany w Pythonie składa się najczęściej z kilku plików tekstowych zawierających *instrukcje* tego języka, z jednym, głównym plikiem *najwyższego*

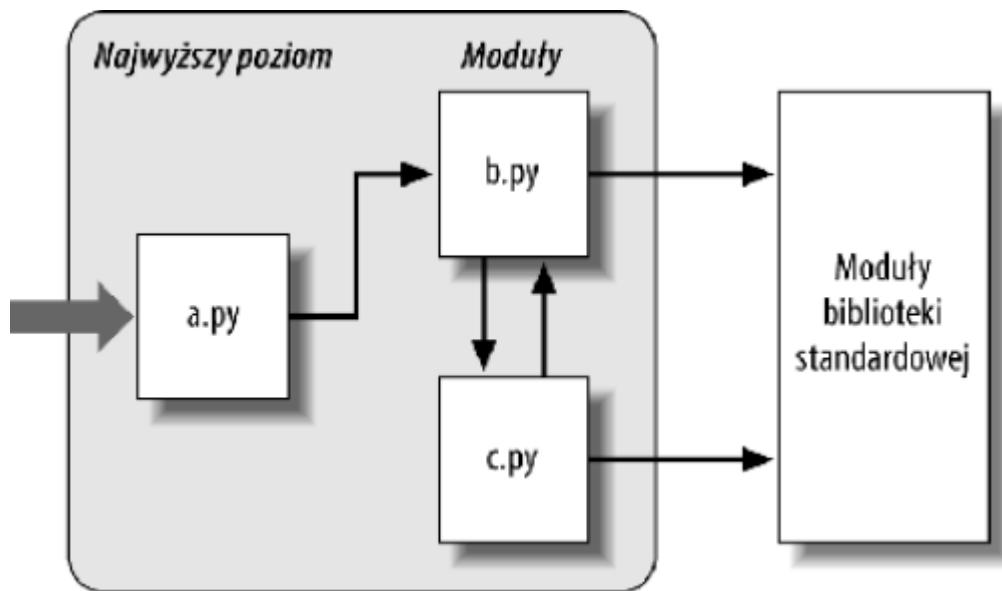
poziomu oraz z zero lub większą liczbą plików pomocniczych, nazywanych w Pythonie *modułami*.

A oto jak to działa. Plik najwyższego poziomu (nazywany także skryptem) zawiera podstawowy przebieg sterowania programu — jest to plik, który wykonuje się w celu uruchomienia aplikacji. Pliki modułów są bibliotekami narzędzi wykorzystywanymi do zebrania komponentów używanych przez plik najwyższego poziomu (i być może również gdzie indziej). Pliki najwyższego poziomu wykorzystują narzędzia zdefiniowane w plikach modułów, a moduły korzystają z narzędzi zdefiniowanych w innych modułach.

Chociaż również zawierają kod programu, to po bezpośrednim uruchomieniu pliki modułów na ogół nie robią nic. Zamiast tego znajdują się w nich definicje narzędzi, które mogą być wykorzystywane w innych plikach. W Pythonie plik *importuje* moduł w celu uzyskania dostępu do narzędzi w nim zdefiniowanych, znanych jako *atrybuty* (czyli nazwy zmiennych dołączone do obiektów, takie jak funkcje). Inaczej mówiąc, importujemy moduły i uzyskujemy dostęp do ich atrybutów w celu wykorzystania ich narzędzi.

Importowanie i atrybuty

Przejdzmy zatem do konkretów. Na rysunku 22.1 przedstawiono strukturę napisanego w Pythonie programu składającego się z trzech plików — *a.py*, *b.py* oraz *c.py*. Plik *a.py* został wybrany jako plik najwyższego poziomu. Będzie prostym plikiem tekstowym z instrukcjami, po uruchomieniu wykonywanym od góry do dołu. Pliki *b.py* oraz *c.py* są modułami. Są również plikami tekstowymi z instrukcjami, jednak zazwyczaj nie są uruchamiane w sposób bezpośredni. Zamiast tego, zgodnie z powyższym opisem, moduły są zazwyczaj importowane przez inne pliki, które chcą korzystać ze zdefiniowanych przez nie narzędzi.



Rysunek 22.1. Architektura programu w Pythonie. Program jest systemem modułów. Zawiera jeden plik skryptu najwyższego poziomu (uruchamiany w celu wykonania programu) i większą liczbę plików modułów (importowanych bibliotek narzędzi). Skrypty oraz moduły są plikami tekstowymi zawierającymi instrukcje Pythona, choć instrukcje z modułów zazwyczaj po prostu tworzą obiekty, które mają być użyte później. Biblioteka standardowa Pythona udostępnia zbiór gotowych modułów

Załóżmy na przykład, że plik *b.py* z rysunku 22.1 definiuje funkcję o nazwie `spam`, która może być wykorzystana przez pliki zewnętrzne. Jak wiemy z omówienia funkcji w czwartej części książki, *b.py* zawiera instrukcję `def`, tworzącą funkcję, którą można później wywołać i przekazać do niej zero lub większą liczbę argumentów, umieszczonych w nawiasach po nazwie funkcji.

```
def spam(text):                                # Plik b.py
    print(text, 'spam')
```

Załóżmy teraz, że skrypt *a.py* chciałby skorzystać z funkcji `spam`. Mógłby zatem zawierać instrukcje podobne do pokazanych poniżej.

```
import b                                     # Plik a.py
b.spam('gumby')                             # Wyświetla tekst "gumby spam"
```

Pierwsza z tych instrukcji, `import`, daje plikowi *a.py* dostęp do wszystkich zmiennych zdefiniowanych przez kod najwyższego poziomu w pliku *b.py*. Polecenie `import b` oznacza mniej więcej tyle:

Załaduj plik *b.py* (o ile nie został on jeszcze załadowany) i daj mi dostęp do wszystkich jego atrybutów przez nazwę `b`.

Aby zrealizować takie zadanie, instrukcje `import` (a także — jak zobaczyłeś później — klauzule `from`) wykonują i ładują inne pliki na żądanie. Bardziej formalnie mówiąc, w Pythonie połączenie między plikami modułów nie jest nawiązane, dopóki nie zostanie wykonana instrukcja taka jak `import`. Rezultatem jej działania jest przypisanie nazw modułów — prostych zmiennych, takich jak `b` — do załadowanych obiektów modułów. Tak naprawdę nazwa modułu użyta w instrukcji `import` spełnia dwie role: identyfikuje ona zewnętrzny *plik*, jaki ma zostać załadowany, ale również staje się *zmienną* przypisaną do załadowanego modułu.

W podobny sposób obiekty *zdefiniowane* w module są również tworzone w czasie wykonywania, kiedy wykonywana jest instrukcja `import`. Instrukcja ta tak naprawdę wykonuje kolejne instrukcje z pliku docelowego w celu utworzenia jego zawartości. Po drodze każda nazwa przypisana na najwyższym poziomie pliku staje się atrybutem modułu, dostępnym dla importerów. Na przykład druga instrukcja z pliku *a.py* wywołuje funkcję `spam` zdefiniowaną w module `b` (i utworzoną poprzez wykonanie polecenia `def` podczas importowania), używając do tego zapisu z atrybutem obiektu. Polecenie `b.spam` oznacza mniej więcej tyle:

Pobierz wartość nazwy `spam` znajdującej się w obiekcie `b`.

W naszym przykładzie nazwa `spam` okazuje się być funkcją wywoływalną, do której w nawiasach przekazujemy ciąg znaków ('`gumby`'). Gdybyś naprawdę utworzył takie pliki, zapisał je na dysku i uruchomił program *a.py*, na ekranie wyświetcone zostałyby słowa `gumby spam`.

Jak z pewnością zauważyleś, notację `obiekt.atrybut` można spotkać praktycznie we wszystkich skryptach napisanych w Pythonie — większość obiektów posiada przydatne atrybuty, które można pobierać za pomocą operatora `..`. Niektóre obiekty wywoływalne są funkcjami, które podejmują określone działania (np. obliczają wysokość prowizji), podczas gdy inne mogą być prostymi wartościami, które oznaczają bardziej statyczne obiekty i właściwości (np. imię czy nazwisko danej osoby).

Pojęcie importowania jest również w Pythonie całkowicie uniwersalne. Dowolny plik może importować narzędzia z każdego innego pliku. Plik *a.py* może na przykład importować *b.py* w celu wywołania jego funkcji, jednak *b.py* może również zaimportować *c.py*, by skorzystać z różnych narzędzi zdefiniowanych w tym module. Łańcuchy importowania mogą sięgać tak głęboko, jak chcemy — w przykładzie wyżej moduł *a* może importować moduł *b*, który importuje moduł *c*, importujący z kolei moduł *b*.

Poza służeniem jako struktura organizacyjna najwyższego poziomu moduły (i pakiety modułów opisane w rozdziale 24.) są również najwyższym poziomem *ponownego wykorzystania kodu* w

Pythonie. Umieszczanie komponentów w plikach modułów sprawia, że są one przydatne w naszym aktualnym programie, ale również w każdym innym programie, jaki możemy kiedyś napisać. Po utworzeniu programu z rysunku 22.1 możemy na przykład odkryć, że funkcja `b.spam` jest uniwersalnym narzędziem, które można wykorzystać w zupełnie innym programie. Wystarczy tylko ponownie zaimportować plik `b.py` w pliku innego programu.

Moduły biblioteki standardowej

Warto zwrócić uwagę na element rysunku 22.1 znajdujący się najbardziej na prawo. Niektóre moduły importowane przez nasz program pochodzą z samego Pythona, a nie z tworzonych przez nas plików.

Python automatycznie zawiera wielki zbiór modułów narzędziowych znany pod nazwą *biblioteki standardowej*. Zbiór ten, składający się z ponad dwustu modułów, zawiera niezależną od platformy obsługę często wykonywanych zadań programistycznych — interfejsów systemu operacyjnego, trwałości obiektów, dopasowywania wzorców tekstowych, skryptów sieciowych i internetowych, tworzenia graficznego interfejsu użytkownika i wiele, wiele innych. Żadne z tych narzędzi nie jest częścią samego języka Python, jednak można z nich korzystać, importując odpowiednie moduły w dowolnej standardowej instalacji Pythona. Ponieważ są to moduły biblioteki standardowej, możesz mieć niemal całkowitą pewność, że będą one dostępne i działające na większości platform, na których uruchamiasz Pythona.

W przykładach z tej książki wykorzystano kilka standardowych modułów biblioteki — na przykład `timeit`, `sys` i `os` w kodzie z poprzedniego rozdziału — ale to tak naprawdę zaledwie niewielki ułamek tego, co nazywamy biblioteką Pythona. Aby uzyskać pełny obraz, powinieneś zapoznać się z dokumentacją biblioteki Pythona, dostępną online na stronie <http://www.python.org> lub dostarczaną wraz z instalacją Pythona (poprzez środowisko IDLE lub menu Start w systemie Windows). Narzędzie *PyDoc*, omówione w rozdziale 15., to kolejny sposób na eksplorację dokumentacji standardowych modułów biblioteki Pythona.

Ponieważ istnieje tak wiele modułów, jest to tak naprawdę jedyny sposób przekonania się, jakie narzędzia są dostępne. Opisy narzędzi biblioteki Pythona można również znaleźć w książkach dotyczących programowania aplikacji, takich jak *Programming Python* mojego autorstwa, jednak dokumentacja jest darmowa, można ją przeglądać w dowolnej przeglądarce internetowej (w formacie HTML), jest dostępna również w innych formatach (np. Windows Help), a co najważniejsze, jest uaktualniana z każdym kolejnym wydaniem Pythona. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 15.

Jak działa importowanie

W poprzednim podrozdziale omawialiśmy importowanie modułów bez objaśnienia, co tak naprawdę się dzieje, kiedy wykonujemy tę operację. Ponieważ importowanie jest w Pythonie sercem struktury programu, w niniejszym podrozdziale przyjrzymy się bardziej formalnym szczegółom operacji importowania, aby proces ten stał się dla nas nieco mniej abstrakcyjny.

Niektórzy programiści języka C lubią porównywać importowanie modułów w Pythonie do dyrektywy `#include` z języka C, jednak tak naprawdę nie powinni tego robić — w Pythonie importowanie nie jest tylko tekstowym wstawieniem jednego pliku do drugiego. Tak naprawdę są to operacje wykonywane po uruchomieniu programu, które przy pierwszym importowaniu danego pliku składają się z trzech osobnych kroków:

1. Odszukanie pliku modułu.
2. Skompilowanie go do postaci kodu bajtowego (jeżeli jest to konieczne).

3. Wykonanie kodu modułu w celu utworzenia definiowanych przez niego obiektów.

Aby lepiej zrozumieć importowanie modułów, omówimy poszczególne kroki po kolei. Pamiętaj, że wszystkie trzy kroki są wykonywane jedynie przy *pierwszym* importowaniu modułu w czasie działania programu. Późniejsze importy tego samego modułu pomijają wszystkie te kroki i po prostu pobierają załadowany obiekt modułu z pamięci. Z technicznego punktu widzenia Python robi to, przechowując załadowane moduły w tabeli o nazwie `sys.modules` i sprawdza jej zawartość na początku operacji importowania. Jeżeli moduł jest tam nieobecny, rozpoczyna się proces importowania składający się z trzech kroków.

1. Odszukanie modułu

Przede wszystkim Python musi zlokalizować plik modułu wskazywany przez daną instrukcję `import`. Warto zauważyć, że instrukcja `import` z przykładu z poprzedniego podrozdziału podaje nazwę pliku bez rozszerzenia `.py` i bez ścieżki do katalogu. W kodzie podano jedynie `import b`, a nie coś w stylu `import c:\katalog1\b.py`. Ścieżka oraz rozszerzenie nazwy pliku pomijane są celowo, ponieważ Python do odszukania pliku modułu wskazywanego w instrukcji `import` wykorzystuje standardową *ścieżkę wyszukiwania modułów* oraz znane typy plików^[1]. Ponieważ jest to najważniejsza część operacji importowania, jaką muszą znać programiści, powrócimy do tego zagadnienia jeszcze raz za moment.

2. Kompilowanie (o ile jest to potrzebne)

Po odnalezieniu w ścieżce wyszukiwania modułów pliku z kodem źródłowym, wskazywanego w instrukcji `import`, Python kompiluje ten plik do postaci kodu bajtowego — jeżeli jest to konieczne. Kod bajtowy omawialiśmy pokrótko w rozdziale 2., ale w rzeczywistości jest to zagadnienie znacznie bardziej złożone. Podczas operacji importowania Python sprawdza zarówno czasy modyfikacji pliku, jak i numer wersji kodu bajtowego, i na tej podstawie podejmuje dalsze decyzje o sposobie działania. Pierwsza operacja wykorzystuje znaczniki czasu pliku zapisane w systemie plików, a druga używa „magicznej” liczby osadzonej w kodzie bajtowym lub nazwy pliku, w zależności od używanej wersji Pythona. W tym kroku wybierana jest jedna z następujących operacji:

Kompilowanie

Jeżeli plik kodu bajtowego jest *starszy* niż plik źródłowy (np. jeżeli kod źródłowy został zmodyfikowany) lub został utworzony przez inną *wersję* języka Python, Python automatycznie generuje kod bajta po uruchomieniu programu.

Jak omówiono wcześniej, ten model został nieco zmodyfikowany w Pythonie 3.2 i późniejszych — pliki bajtów są segregowane w podkatalogu `_pycache_` i nazywane wraz z wersją Pythona, aby uniknąć rywalizacji i ponownej komplikacji, gdy zainstalowanych jest wiele Pythonów. Eliminuje to potrzebę sprawdzania numerów wersji w kodzie bajtowym, ale sprawdzanie znacznika czasu jest nadal używane do wykrywania zmian w źródle.

Brak kompilowania

Jeżeli jednak okaże się, że plik z kodem bajtowym `.pyc` *nie jest* starszy od odpowiadającego mu pliku z kodem źródłowym i został utworzony przy użyciu tej samej wersji Pythona, komplikacja kodu źródłowego na kod bajtowy jest pomijana.

Dodatkowo jeżeli Python odnajduje w ścieżce wyszukiwania jedynie plik z kodem bajtowym, a nie widzi źródeł, po prostu ładuje kod bajtowy bezpośrednio (co oznacza, że możemy publikować program w postaci samych plików z kodem bajtowym i nie udostępniać plików źródłowych). Innymi słowy, jeżeli jest to możliwe, komplikacja jest *pomijana* w celu przyspieszenia uruchamiania programu.

Warto zauważyć, że komplikacja ma miejsce, kiedy plik jest importowany. Z tego powodu zazwyczaj nie widzimy kodu pliku bajtowego `.pyc` dla pliku *najwyższego poziomu* programu, o ile nie zostanie on zimportowany przez inny plik — jedynie pliki importowane pozostawiają na naszym komputerze pliki `.pyc`. Kod bajtowy plików najwyższego poziomu jest zazwyczaj wykorzystywany wewnętrznie i usuwany. Kod bajtowy importowanych plików jest zapisywany w plikach, by móc w przyszłości poprawić szybkość działania przyszłych importów.

Pliki najwyższego poziomu są często zaprojektowane do bezpośredniego wykonywania, a nie importowania. Później zobaczymy, że można projektować pliki służące zarówno jako kod najwyższego poziomu programu, jak i moduł narzędzi do zimportowania. Takie pliki mogą być wykonywane oraz importowane i tym samym generują pliki `.pyc`. Aby przekonać się, jak to działa, powinieneś zatrzymać się na etapie omówienia specjalnych atrybutów `_name_` oraz `_main_` w rozdziale 25.

3. Wykonanie

Ostatnim krokiem w operacji importowania jest wykonanie kodu bajtowego modułu. Wszystkie instrukcje pliku są wykonywane po kolejno, od góry do dołu, a przypisania do nazw wykonane na tym etapie generują atrybuty wynikowego obiektu modułu. W ten sposób tworzone są narzędzia zdefiniowane w kodzie modułu. Na przykład instrukcje `def` pliku są wykonywane w czasie importowania w celu utworzenia funkcji i przypisania atrybutów wewnętrznych modułu do tych funkcji. Funkcje mogą następnie być wywoływane później w programie przez kod importujący plik.

Ponieważ ostatni krok operacji importowania naprawdę wykonuje kod pliku, to w przypadku kiedy kod najwyższego poziomu w pliku modułu realizuje jakieś określone zadanie, zobaczymy jego wynik w czasie importowania. Na przykład instrukcje `print` na najwyższym poziomie modułu po zimportowaniu pliku wyświetlały wyniki działania. Instrukcje `def` funkcji z kolei po prostu definiują obiekty, które mogą zostać wykorzystane później.

Jak widać, operacje importowania obejmują całkiem sporo pracy — wyszukują pliki, mogą komplikować kod źródłowy i wykonują kod Pythona. Z tego powodu każdy moduł importowany jest domyślnie tylko raz na proces. Kolejne operacje importowania pomijają wszystkie trzy etapy i korzystają z modułu już załadowanego do pamięci. Jeżeli chcemy ponownie zimportować plik po tym, jak został już załadowany (na przykład w celu dynamicznego dostosowania kodu do potrzeb użytkownika), musimy to wymusić za pomocą wywołania funkcji `imp.reload` — narzędzia, z którym spotkamy się w kolejnym rozdziale [2].

Pliki kodu bajtowego — `_pycache_` w Pythonie 3.2+

Jak już wcześniej krótko wspominaliśmy, sposób, w jaki Python przechowuje pliki w celu zachowania kodu bajtowego utworzonego przez skompilowanie kodu źródłowego, zmienił się w Pythonie 3.2 i nowszych. Po pierwsze, jeżeli Python z jakiegoś powodu nie może zapisać pliku kodu bajtowego na dysku, program nadal będzie działał poprawnie — Python po prostu tworzy kod bajtowy w pamięci, korzysta z niego podczas działania programu i usuwa go po zakończeniu. Aby przyspieszyć uruchamianie, po utworzeniu kodu bajtowego Python próbuje zapisać go w pliku na dysku, aby następnym razem pominąć krok komplikacji. Sposób, w jaki się to odbywa, zależy od wersji Pythona:

W Pythonie 3.1 i wersjach wcześniejszych (w tym wszystkich z serii Python 2.x)

Kod bajtowy jest przechowywany w plikach *w tym samym katalogu*, co odpowiadające mu pliki źródłowe, zwykle z rozszerzeniem nazwy *.pyc* (np. *module.pyc*). Pliki kodu bajtowego są również wewnętrznie znakowane wersją Pythona, za pomocą której zostały utworzone (programiści znają to jako tzw. „magiczną liczbę”), dzięki czemu Python wie, że powinien ponownie skompilować kod źródłowy, kiedy oznaczenie wersji Pythona w pliku kodu bajtowego różni się od aktualnie używanej wersji Pythona. Na przykład, jeżeli zaktualizujesz swojego Pythona do najnowszej wersji, której „magiczna liczba” różni się od dotychczasowej, wszystkie pliki kodu bajtowego zostaną ponownie automatycznie skompilowane z powodu niezgodności numeru wersji, nawet jeżeli odpowiadający mu kod źródłowy nie został zmodyfikowany.

W Pythonie 3.2 i nowszych wersjach

Kod bajtowy jest przechowywany w plikach w podkatalogu o nazwie *_pycache_*, który Python tworzy w razie potrzeby i który znajduje się w katalogu zawierającym odpowiednie pliki źródłowe. Pomaga to uniknąć *bałaganu* w katalogach źródłowych, ponieważ pliki kodu bajtowego są przechowywane w osobnym, własnym katalogu. Ponadto mimo że pliki kodu bajtowego nadal mają rozszerzenie *.pyc*, jak poprzednio, otrzymują bardziej opisowe nazwy zawierające tekst identyfikujący wersję Pythona, która je utworzyła (np. *module.cpython-32.pyc*). Pozwala to uniknąć konfliktów i *rekompilacji* (ponownej komplikacji) kodu, ponieważ każda wersja zainstalowanego Pythona może mieć własną, unikatowo nazwaną wersję plików kodu bajtowego w podkatalogu *_pycache_* — dzięki takiemu rozwiązaniu nie ma potrzeby zastępowania kodu bajtowego nową wersją ani jego ponownej komplikacji. Technicznie rzecz biorąc, nazwy plików kodu bajtowego zawierają także *nazwę Pythona*, który je utworzył, zatem CPython, Jython i inne implementacje wspomniane we wstępnie i rozdziale 2. mogą współistnieć na tym samym komputerze bez wchodzenia sobie w paradę (o ile obsługują już taki model działania).

W *obu* modelach Python zawsze odtwarza plik kodu bajtowego, jeżeli kod źródłowy został zmodyfikowany od czasu ostatniej komplikacji, ale różnice w wersjach są obsługiwane w inny sposób — według „magicznych liczb” i zastępowania plików przed wersją 3.2 oraz według nazw plików, które pozwalają na wiele kopii kodu bajtowego w wersji 3.2 i nowszych.

Modele plików kodu bajtowego w akcji

Poniżej znajduje się szybki przykład tych dwóch modeli działających w wersjach 2.x i 3.3. Pominąłem tutaj dużo tekstu wyświetlanego przez polecenie *dir* w systemie Windows ze względu na oszczędność miejsca, a zastosowany tu skrypt nie został pokazany, ponieważ nie jest istotny w tym omówieniu (pochodzi z rozdziału 2. i po prostu wyświetla dwie wartości). W wersjach wcześniejszych niż 3.2 pliki kodu bajtowego pojawiają się obok plików źródłowych po utworzeniu w czasie operacji importowania:

```
c:\code\py2x> dir
10/31/2012  10:58 AM              39 script0.py
c:\code\py2x> C:\python27\python
>>> import script0
hello world
1267650600228229401496703205376
>>> ^Z
c:\code\py2x> dir
10/31/2012  10:58 AM              39 script0.py
```

```
10/31/2012 11:00 AM           154 script0.pyc
```

Jednak w wersji 3.2 i nowszych pliki kodu bajtowego są zapisywane w podkatalogu `__pycache__` i zawierają w swoich nazwach wersje i nazwy implementacji Pythona, aby uniknąć bałaganu i konfliktów między Pythonami działającymi na danym komputerze:

```
c:\code\py2x> cd ..\py3x
c:\code\py3x> dir
10/31/2012 10:58 AM           39 script0.py

c:\code\py3x> C:\python33\python
>>> import script0
hello world
1267650600228229401496703205376
>>> ^Z

c:\code\py3x> dir
10/31/2012 10:58 AM           39 script0.py
10/31/2012 11:00 AM    <DIR>          __pycache__
c:\code\py3x> dir __pycache__
10/31/2012 11:00 AM           184 script0.cpython-33.pyc
```

Co najważniejsze, w modelu używanym w Pythonie 3.2 i wersjach późniejszych importowanie tego samego pliku za pomocą innej wersji Pythona tworzy inny plik kodu bajtowego, zamiast nadpisywania istniejącego, pojedynczego pliku, jak to miało miejsce w wersjach wcześniejszych niż 3.2 — w nowszym modelu każda wersja Pythona ma swoje własne pliki kodu bajtowego, gotowe do załadowania przy następnym uruchomieniu programu (choć wcześniejsze Pythony będą kontynuować użytkowanie swojego schematu na tym samym komputerze):

```
C:\code\py3x> C:\python32\python
>>> import script0
witaj, świecie
1267650600228229401496703205376
>>> ^Z

c:\code\py3x> dir __pycache__
10/31/2012 12:28 PM           178      script0.cpython-32.pyc
10/31/2012 11:00 AM           184      script0.cpython-33.pyc
```

Nowszy model pliku kodu bajtowego z wersji Python 3.2 jest prawdopodobnie lepszy, ponieważ pozwala uniknąć ponownej komplikacji, gdy na komputerze znajduje się więcej niż jedna wersja Pythona — częsty przypadek w dzisiejszym mieszanym świecie wersji 2.x/3.x. Z drugiej strony taki model nie jest pozbawiony potencjalnych niezgodności w programach opartych na wcześniejszej strukturze plików i katalogów. Może to być na przykład problem kompatybilności w niektórych programach narzędziowych, chociaż większość dobrze napisanych narzędzi powinna działać tak jak poprzednio. Więcej szczegółowych informacji na temat potencjalnych skutków wynikających ze zmiany modelu pliku kodu bajtowego znajdziesz w dokumencie *What's New?* (co nowego?) w Pythonie 3.2.

Pamiętaj również, że proces ten jest całkowicie *automatyczny* — jest to efekt uboczny uruchamiania programów — stąd większość programistów prawdopodobnie nie będzie dbać o te różnice, a nawet ich nie zauważą, może poza szybszymi uruchomieniami z powodu mniejszej liczby ponownych komplikacji.

Ścieżka wyszukiwania modułów

Jak wspominaliśmy wcześniej, najistotniejszą dla programistów częścią procedury importowania jest zazwyczaj część pierwsza — lokalizacja pliku, który ma zostać zimportowany (część „*odnalezienie*”). Ponieważ możesz być zmuszony do poinformowania Pythona o tym, gdzie ma szukać importowanych plików, musisz wiedzieć, w jaki sposób można dostać się do jego ścieżki wyszukiwania i jak ją rozszerzyć.

W wielu przypadkach możemy polegać na automatycznej naturze ścieżki wyszukiwania importowanych modułów i nie musimy jej wcale konfigurować. Jeżeli jednak chcesz importować pliki ponad granicami katalogów, musisz wiedzieć, w jaki sposób działa ścieżka wyszukiwania, by móc ją dostosować do własnych potrzeb. Ścieżka wyszukiwania modułów Pythona składa się z zestawienia poniższych komponentów podstawowych. Niektóre z nich są ustawione za nas, inne musimy dostosować do własnych potrzeb, by przekazać Pythonowi, gdzie ma szukać.

1. Katalog główny programu.
2. Katalogi PYTHONPATH (jeżeli są ustawione).
3. Katalogi biblioteki standardowej.
4. Zawartość wszystkich plików *.pth* (jeżeli są one obecne).
5. Katalog *site-packages* dla zewnętrznych pakietów rozszerzeń.

Zestawienie tych pięciu komponentów daje nam `sys.path` — mutowalną listę nazw katalogów, o której powiemy nieco więcej w dalszej części tego podrozdziału. Pierwszy i trzeci element ścieżki wyszukiwania definiowane są automatycznie. Ponieważ jednak Python przeszukuje zestawienie tych komponentów od pierwszego do ostatniego, *drugi* i *czwarty* element można wykorzystać do rozszerzenia tej ścieżki w taki sposób, aby obejmowała ona nasze własne katalogi z kodem źródłowym. Poniżej przedstawiono, w jaki sposób Python wykorzystuje każdy z tych komponentów ścieżki.

Katalog główny (*automatyczne*)

Python najpierw szuka importowanych plików w katalogu głównym. Znaczenie „katalogu głównego” będzie różne w zależności od sposobu uruchamiania kodu. Jeżeli wykonujemy program, będzie to katalog zawierający plik skryptowy najwyższego poziomu tego programu. Jeżeli pracujemy *interaktywnie*, będzie to katalog, w którym pracujemy (czyli aktualny katalog roboczy).

Ponieważ ten katalog zawsze przeszukiwany jest jako pierwszy, jeśli program jest w całości umieszczony w jednym katalogu, wszystkie operacje importowania będą działać automatycznie, bez konieczności konfigurowania ścieżki. Z drugiej strony, ponieważ katalog ten przeszukiwany jest jako pierwszy, jego pliki będą także przeciążały moduły o tej samej nazwie znajdujące się w katalogach podanych w innych miejscach ścieżki. Uważaj, aby w ten sposób nie ukryć przypadkowo modułów biblioteki, jeśli będą Ci one potrzebne w programie, lub skorzystaj z narzędzi pakietów, które mogą częściowo ominąć ten problem (opowiem o nich już niebawem).

Katalogi PYTHONPATH (*konfigurowalne*)

Następnie Python przeszukuje wszystkie katalogi podane w zmiennej środowiskowej `PYTHONPATH`, od lewej do prawej strony (zakładając, że ją w ogóle ustawiliśmy; ta zmienna nie jest predefiniowana). Mówiąc w skrócie, w `PYTHONPATH` podaje się listę zdefiniowanych przez użytkownika i specyficznych dla platformy nazw katalogów zawierających pliki z kodem Pythona. Możemy tam dodać wszystkie katalogi, z których chcemy coś importować, a Python rozszerzy ścieżkę wyszukiwania modułów w taki sposób, aby obejmowała ona wszystkie katalogi podane w `PYTHONPATH`.

Ponieważ Python najpierw przeszukuje katalog główny, to ustawienie ma znaczenie jedynie wtedy, gdy importuje się pliki pomiędzy różnymi katalogami — to znaczy, kiedy musimy zainportować plik przechowywany w katalogu *innym* niż katalog pliku go importującego. Najprawdopodobniej będziesz chciał ustawić zmienną `PYTHONPATH`, kiedy zaczniesz pisać większe programy, jednak na początku, dopóki zapisujesz wszystkie pliki modułów w katalogu, w którym pracujesz (na przykład katalogu `C:\code`, którego używamy w tej książce), operacje importowania powinny działać bez konieczności martwienia się o to ustawienie.

Katalogi biblioteki standardowej (automatyczne)

Następnie Python automatycznie przeszukuje katalogi, w których na komputerze zainstalowane są moduły biblioteki standardowej. Ponieważ są one przeszukiwane zawsze, zazwyczaj nie musisz martwić się o dodanie ich do zmiennej środowiskowej `PYTHONPATH` lub do omówionych niżej plików ścieżek.

Katalogi ścieżek plików .pth (konfigurowalne)

Kolejna opcja jest stosunkowo rzadziej wykorzystywana, choć pozwala użytkownikom dodawać katalogi do ścieżki wyszukiwania modułów przez proste umieszczenie ich po jednym w wierszu w pliku tekstowym kończącym się rozszerzeniem `.pth` (od ang. `path` — ścieżka). Takie pliki konfiguracyjne są raczej zaawansowaną opcją instalacyjną, dlatego nie będziemy ich tutaj w pełni omawiać. Stanowią one alternatywę dla ustawień zmiennej środowiskowej `PYTHONPATH`.

W skrócie, pliki tekstowe z nazwami katalogów wstawione do odpowiedniego katalogu mogą pełnić tę samą rolę co zmienna środowiskowa `PYTHONPATH`. Na przykład, jeżeli korzystasz z systemu Windows i Pythona 3.3, ścieżkę wyszukiwania modułów możesz w prosty sposób rozszerzyć, umieszczając plik o nazwie `myconfig.pth` w głównym katalogu instalacyjnym Pythona (`C:\Python33`) lub w podkatalogu pakietów biblioteki standardowej (`C:\Python33\Lib\site-packages`). W systemach uniksowych plik ten można umieścić na przykład w katalogu `/usr/local/lib/python3.3/site-packages` lub `/usr/local/lib/site-python`.

Kiedy plik taki jest obecny, Python dodaje katalogi wymienione w wierszach pliku, od pierwszego do ostatniego, na końcu listy ścieżki wyszukiwania modułów — w obecnej wersji będzie to po `PYTHONPATH` i bibliotekach standardowych, ale przed katalogiem `site-packages`, gdzie często instalowane są rozszerzenia innych firm. Tak naprawdę Python zbiera nazwy katalogów ze wszystkich plików `.pth`, jakie znajdzie, a następnie odfiltrowuje wszystkie powtarzające się lub nieistniejące katalogi. Ponieważ są to pliki, a nie ustawienia powłoki, pliki ze ścieżkami mogą mieć zastosowanie dla wszystkich użytkowników danej instalacji, a nie tylko dla jednego użytkownika lub powłoki. Co więcej, dla niektórych użytkowników lub aplikacji utworzenie takich plików tekstowych może być łatwiejsze od użycia zmiennych środowiskowych.

Opcja ta jest nieco bardziej zaawansowana, niż wskazywałby na to powyższy opis. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona (w szczególności dotyczącej modułu `site` — moduł ten pozwala na konfigurację lokalizacji bibliotek Pythona oraz plików ścieżek, a w jego dokumentacji opisano także oczekiwane lokalizacje plików ścieżek). Osobom początkującym polecam korzystanie z `PYTHONPATH` lub być może pojedynczego pliku `.pth`, i to tylko wtedy, gdy muszą one importować moduły pomiędzy różnymi katalogami. Pliki ze ścieżkami są częściej wykorzystywane przez

biblioteki zewnętrzne, które najczęściej instalują pliki tego typu w katalogu *site-packages*, o którym opowiem w kolejnym podpunkcie.

Katalog Lib\site-packages dla zewnętrznych pakietów rozszerzeń (automatyczne)

Wreszcie na koniec Python automatycznie dodaje podkatalog *site-packages* swojej biblioteki standardowej do ścieżki wyszukiwania modułów. Umownie jest to miejsce, w którym instaluje się większość rozszerzeń innych firm, często automatycznie przez narzędzie *distutils*, opisane w ramce w dalszej części tego rozdziału. Ponieważ ich katalog instalacyjny jest zawsze częścią ścieżki wyszukiwania modułów, klienci mogą importować moduły takich rozszerzeń bez konieczności osobnego ustawiania ścieżki.

Konfigurowanie ścieżki wyszukiwania

W rezultacie zarówno PYTHONPATH, jak i komponenty plików *.pth* ze ścieżki wyszukiwania pozwalają dostosować miejsca wyszukiwania importowanych plików do własnych potrzeb. Sposób ustawiania zmiennych środowiskowych i miejsce przechowywania plików ścieżek różni się dla poszczególnych platform. Na przykład w systemie Windows można wykorzystać ikonę *System z Panelu sterowania* do zapisania w zmiennej PYTHONPATH listy katalogów rozdzielonych od siebie średnikami, tak jak to zostało pokazane poniżej:

```
c:\pycode\utilities;d:\pycode\package1
```

Można również zamiast tego utworzyć plik tekstowy o nazwie *C:\Python33\pydirs.pth* i następującej zawartości:

```
c:\pycode\utilities  
d:\pycode\package1
```

Ustawienia te na innych platformach wyglądają analogicznie, jednak szczegóły mogą się od siebie różnić w zbyt dużym stopniu, abyśmy opisywali je tutaj w całości. Przykłady często spotykanych sposobów rozszerzania ścieżki wyszukiwania modułów za pomocą PYTHONPATH oraz plików *.pth* na różnych platformach można znaleźć w dodatku A.

Wariacje ścieżki wyszukiwania modułów

Powyższy opis ścieżki wyszukiwania modułów jest poprawny, ale bardzo ogólny. Dokładny sposób konfiguracji ścieżki wyszukiwania może się różnić na innych platformach i dla różnych wersji oraz dystrybucji Pythona. W zależności od platformy dodatkowe katalogi mogą być również automatycznie dodawane do ścieżki wyszukiwania modułów.

Na przykład niektóre wersje Pythona mogą dodawać wpis dla *bieżącego katalogu roboczego* — katalogu, z którego uruchomiliśmy program — do ścieżki wyszukiwania przed katalogami ze zmiennej PYTHONPATH. Po uruchomieniu z poziomu wiersza poleceń bieżący katalog roboczy może nie być tym samym co katalog roboczy uruchamianego pliku najwyższego poziomu (czyli katalog, w którym znajduje się plik programu), który jest zawsze dodawany. Ponieważ bieżący katalog roboczy może zmieniać się za każdym wykonaniem programu, zazwyczaj nie powinniśmy polegać na jego wartości na potrzeby importowania. Więcej informacji na temat uruchamiania programów z wiersza poleceń znajdziesz w rozdziale 3.[\[3\]](#)

Aby sprawdzić, jak Python konfiguruje ścieżkę wyszukiwania modułów dla określonej platformy, zawsze możesz zbadać listy *sys.path* — to zagadnienie zostało omówione poniżej.

Lista sys.path

Jeżeli chcesz sprawdzić, jak na Twoim komputerze skonfigurowana jest ścieżka wyszukiwania modułów, zawsze możesz to zobaczyć w takiej samej postaci, jak widzi to Python, wyświetlając wbudowaną listę `sys.path` (czyli atrybut `path` modułu `sys` z biblioteki standardowej). Ta lista nazw katalogów jest prawdziwą ścieżką wyszukiwania w Pythonie. W przypadku operacji importowania Python przeszukuje każdy katalog listy od lewej do prawej strony i wykorzystuje pierwszy pasujący element.

Tak naprawdę `sys.path` jest ścieżką wyszukiwania modułów. Python konfiguruje ją w momencie uruchomienia programu, automatycznie łącząc w listę katalog główny pliku najwyższego poziomu (lub pusty łańcuch znaków oznaczający bieżący katalog roboczy), wszystkie katalogi z `PYTHONPATH`, zawartość wszelkich utworzonych plików `.pth`, a także katalogi biblioteki standardowej. W rezultacie otrzymujemy listę łańcuchów znaków nazw katalogów, którą Python przeszukuje przy każdej operacji importowania nowego pliku.

Python udostępnia tę listę z dwóch ważnych powodów. Po pierwsze, jest to sposób pozwalający na zweryfikowanie wprowadzonych ustawień ścieżki — jeżeli Twoich ustawień na tej liście nie będzie, będziesz musiał sprawdzić, czy wszystko zrobiłeś poprawnie. Przykładowo poniżej pokazujemy, jak wygląda moja ścieżka wyszukiwania modułów w systemie Windows, dla Pythona 3.3, ze zmienią średowiskową `PYTHONPATH` ustawioną na `C:\code` oraz plikiem ścieżki `C:\Python33\mypath.py` zawierającym katalog `C:\users\mark`. Pusty ciąg znaków na początku oznacza, że połączony został katalog bieżący oraz moje dwa ustawienia; reszta to katalogi i pliki biblioteki standardowej oraz katalog `site-packages` dla rozszerzeń zewnętrznych:

```
>>> import sys  
>>> sys.path  
[ '', 'C:\\\\code', 'C:\\Windows\\\\system32\\\\python33.zip', 'C:\\\\Python33\\\\DLLs',  
 'C:\\\\Python33\\\\lib', 'C:\\\\Python33', 'C:\\\\Users\\\\mark',  
 'C:\\\\Python33\\\\lib\\\\site-packages' ]
```

Po drugie, jeżeli wiesz, co robisz, taka lista zapewnia również sposób na ręczne dostosowywanie ścieżek wyszukiwania dla skryptów. Jak zobaczymy później w dalszej części książki, modyfikując `sys.path`, możemy również modyfikować ścieżkę wyszukiwania dla wszystkich przyszłych importów. Takie zmiany trwają jednak tylko do końca czasu działania skryptu; zmenna `PYTHONPATH` i pliki `.pth` oferują bardziej trwałe sposoby modyfikowania tej ścieżki — pierwszy z nich dla użytkownika, a drugi dla instalacji.

Z drugiej strony niektóre programy naprawdę muszą zmienić zawartość listy `sys.path`. Na przykład skrypty uruchamiane na serwerach WWW często działają jako użytkownik `nobody`, aby ograniczyć dostęp do komputera. Ponieważ takie skrypty zwykle nie mogą polegać na tym, że ten użytkownik będzie ustawał zmenną `PYTHONPATH` w jakiś konkretny sposób, przed uruchomieniem jakichkolwiek instrukcji importu zazwyczaj ustawiają listę `sys.path` ręcznie, aby uwzględnić wymagane katalogi źródłowe. Często wystarcza użycie metody `sys.path.append` lub `sys.path.insert`, choć takie rozwiązanie przetrwa tylko jedno uruchomienie programu.

Wybór pliku modułu

Pamiętaj, że rozszerzenia plików (na przykład `.py`) są w instrukcjach `import` celowo pomijane. Python wybiera pierwszy plik odpowiadający importowanej nazwie, jaki znajdzie w ścieżce wyszukiwania. W rzeczywistości operacje importu są punktami interfejsu do wielu zewnętrznych komponentów — kodu źródłowego, wielu odmian kodu bajtowego, skompilowanych rozszerzeń i innych. Python automatycznie wybiera dowolny typ pasujący do nazwy modułu.

Kody źródłowe modułów

Na przykład instrukcja `import b` może obecnie załadować lub zostać rozwiązana jako:

- Plik kodu źródłowego o nazwie `b.py`.
- Plik kodu bajtowego nazwie `b.pyc`.
- Plik zoptymalizowanego kodu bajtowego o nazwie `b.pyo` (nieco mniej popularny format).
- Katalog o nazwie `b` dla importowania pakietów (opisanego w rozdziale 24.).
- Skompilowany moduł rozszerzenia, zazwyczaj napisany w językach C, C++ lub innych, dynamicznie dołączany w momencie importowania (na przykład `b.so` w systemie Linux, `b.dll` lub `b.pyd` dla środowisk Cygwin i Windows).
- Skompilowany moduł wbudowany napisany w języku C i statycznie dołączony do Pythona.
- Komponent pliku ZIP automatycznie rozpakowywany po zainportowaniu.
- Obraz obszaru pamięci, dla zamrożonych plików wykonywalnych.
- Klasa języka Java dla dystrybucji Jython języka Python.
- Komponent .NET dla dystrybucji IronPython języka Python.

Rozszerzenia języka C, Jython oraz importowanie pakietów rozszerzają operacje importowania poza proste pliki. Dla kodu importującego różnice w typach załadowanych plików są jednak zupełnie nieistotne, zarówno przy importowaniu, jak i przy pobieraniu atrybutów modułów. Instrukcja `import b` pobiera dowolny moduł `b`, czymkolwiek by on był, zgodnie ze ścieżką wyszukiwania modułów, natomiast `b.atrybut` pobiera element z tego modułu — obojętnie, czy jest to zmienna Pythona, czy dołączona funkcja języka C. Niektóre moduły biblioteki standardowej, z których będziemy korzystać w książce, są tak naprawdę napisane w języku C, a nie w Pythonie; jednak ze względu na fakt, że dla kodu importującego wyglądają jak pliki modułów w języku Python, nie ma to żadnego znaczenia.

Priorytety wyboru

Jeżeli pliki `b.py` i `b.so` znajdują się w różnych katalogach, Python zawsze załada ten znaleziony w pierwszym (znajującym się bardziej po lewej stronie) katalogu ścieżki wyszukiwania modułów w czasie przeszukiwania listy `sys.path` od lewej do prawej strony. Co się jednak dzieje, jeżeli Python znajdzie zarówno `b.py`, jak i `b.so` w tym samym katalogu? W takim przypadku Python postępuje zgodnie ze standardową kolejnością wybierania, choć nie ma żadnej gwarancji, że kolejność ta nie zmieni się w innych wersjach lub dystrybucjach. Nie powinieneś zatem polegać na tym, który rodzaj pliku o tej samej nazwie w danym katalogu wybierze Python — zamiast tego lepiej jest nadawać plikom różne nazwy lub skonfigurować ścieżkę wyszukiwania modułów w taki sposób, by nasze preferencje w zakresie wybierania modułów były bardziej oczywiste.

Importowanie punktów zaczepienia i plików ZIP

Normalnie importowanie działa tak, jak opisano to w niniejszym podrozdziale — Python odszukuje i ładuje pliki zgodnie z regułami. W razie potrzeby możemy jednak przynajmniej częściowo redefiniować w Pythonie sposób działania operacji importowania, używając do tego celu tak zwanych *punktów zaczepienia operacji importowania* (ang. *import hooks*). Takie punkty zaczepienia można wykorzystać do wykonywania podczas importowania wielu przydatnych operacji, takich jak na przykład ładowanie plików z archiwów czy odszyfrowywanie danych i innych.

Tak naprawdę sam Python wykorzystuje punkty zaczepienia w celu umożliwienia bezpośredniego importu plików z archiwów ZIP — zarchiwizowane pliki są automatycznie pobierane w czasie importowania, kiedy plik `.zip` zostanie wybrany w ścieżce wyszukiwania importowanych modułów. Jeden z katalogów biblioteki standardowej w powyższym kodzie wyświetlającym zawartość `sys.path` był na przykład plikiem `.zip`. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona, w opisie wbudowanej funkcji `_import_` — narzędzia, które można dostosowywać do własnych potrzeb, a które tak naprawdę wykonywane jest przez instrukcję `import`.

Więcej szczegółowych informacji na temat omawianych tutaj zagadnień



znajdziesz w dokumencie *What's New?* (co nowego) Pythona 3.3; wiele z tych informacji zostało tutaj celowo pominiętych ze względu na brak miejsca. Krótko mówiąc, w wersji 3.3 i nowszych funkcja `__import__` jest zaimplementowana poprzez `importlib.__import__`, częściowo w celu ujednolicenia i lepszego wyekspresowania sposobu jej wdrożenia.

Ostatnie z tych wywołań jest również opakowane w `importlib.import_module` — narzędzie, które zgodnie z aktualnymi podręcznikami Pythona jest ogólnie preferowane zamiast `__import__` do bezpośrednich wywołań importu według nazw; technika ta została omówiona w rozdziale 25. Oba wywołania nadal działają, chociaż funkcja `__import__` obsługuje dostosowywanie importu przez zastąpienie we wbudowanym zasięgu (patrz rozdział 17.), a inne rozwiązania obsługują podobne role.Więcej informacji znajdziesz w dokumentacji bibliotek Pythona.

Oprogramowanie zewnętrzne — `distutils`

Przedstawiony w tym rozdziale opis ustawień ścieżki wyszukiwania modułów odnosi się przede wszystkim do kodu źródłowego pisanego przez użytkownika. Rozszerzenia zewnętrzne dla Pythona zazwyczaj wykorzystują do automatycznej instalacji narzędzia takie jak `distutils` z biblioteki standardowej, dzięki czemu nie jest wymagane dodatkowe konfigurowanie ścieżek, aby używać ich kodu.

Systemy wykorzystujące `distutils` najczęściej są dostarczane ze skryptem `setup.py`, który jest uruchamiany w celu ich zainstalowania. Skrypt ten importuje i wykorzystuje moduły `distutils` do umieszczenia instalowanego pakietu w katalogu stojącym się automatycznie częścią ścieżki wyszukiwania modułów (zazwyczaj w podkatalogu `Lib\site-packages` Pythona, niezależnie od tego, gdzie on się znajduje).

Więcej informacji dotyczących dystrybucji i instalacji za pomocą `distutils` możesz znaleźć w dokumentacji biblioteki standardowej Pythona. Wykorzystywanie tych narzędzi pozostaje poza zakresem niniejszej książki (pozwalały one na przykład automatycznie kompilować rozszerzenia napisane w języku C na komputerze docelowym). Powinieneś również zapoznać się z zewnętrznym systemem typu open source o nazwie `eggs`, który dodaje mechanizm sprawdzania zależności dla zainstalowanego oprogramowania Pythona.

Uwaga: kiedy pracowałem na piątym wydaniu tej książki, zaczęły się pojawiać pogłoski o wycofywaniu pakietu `distutils` i zastąpieniu go w standardowej bibliotece Pythona nowszym pakietem `distutils2`. Status tego pakietu nie jest jeszcze jasny — oczekiwano go w wersji 3.3, ale się nie pojawił, dlatego zawsze powinieneś zaglądać do plików *What's New?* kolejnych wersji Pythona, aby uzyskać informacje o wprowadzanych aktualizacjach, które mogą pojawić się po wydaniu tej książki.

Pliki zoptymalizowanego kodu bajtowego

Python obsługuje również pliki zoptymalizowanego kodu bajtowego (`.pyo`), tworzone i wykonywane poprzez umieszczenie w wierszu polecenia opcji `-O` i automatycznie generowane przez niektóre narzędzia instalacyjne. Ponieważ działają one tylko odrobinę szybciej od normalnych plików `.pyc` (zazwyczaj jest to różnica około pięciu procent), nie są używane zbyt często. Warto zauważyć, że na przykład system PyPy (opisany w rozdziale 2. i rozdziale 21.) umożliwia znacznie większe przyspieszenie działania kodu.Więcej szczegółowych informacji na temat plików `.pyo` znajdziesz w dodatku A i rozdziale 36.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy podstawy modułów, atrybutów oraz importowania, a także działanie instrukcji `import`. Zobaczyliśmy, że operacje importowania odnajdują plik docelowy w

ścieżce wyszukiwania modułów, komplują go do postaci kodu bajtowego i wykonują wszystkie jego instrukcje w celu wygenerowania zawartości. Dowiedzieliśmy się również, w jaki sposób — przede wszystkim za pomocą ustawień zmiennej środowiskowej `PYTHONPATH` — konfiguruje się ścieżkę wyszukiwania, tak by móc importować z innych katalogów poza katalogiem głównym oraz katalogami biblioteki standardowej.

Jak pokazał niniejszy rozdział, operacja importowania i moduły są sercem architektury programów Pythona. Większe programy podzielone są na większą liczbę plików łączonych razem przez operacje importowania w czasie wykonywania. Importowanie wykorzystuje z kolei ścieżkę wyszukiwania modułów do lokalizacji plików, a moduły definiują atrybuty do wykorzystania przez pliki zewnętrzne.

Oczywiście cały cel importowania oraz modułów polega na nadaniu programom struktury dzielącej ich logikę na samodzielne komponenty oprogramowania. Kod z jednego modułu jest izolowany od kodu z pozostałych modułów. Tak naprawdę żaden plik nie może zobaczyć nazw zdefiniowanych w innym, dopóki w jawnym sposobie nie wykona się instrukcji `import`. Z tego powodu moduły minimalizują konflikty pomiędzy różnymi częściami programu w zakresie nazw zmiennych.

W kolejnym rozdziale przekonamy się, co to wszystko znaczy w praktyce, na przykładzie prawdziwego kodu. Zanim jednak przejdziemy dalej, zajmijmy się quizem dotyczącym niniejszego rozdziału.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób plik z kodem źródłowym modułu staje się obiektem modułu?
2. Po co ustawiamy zmienną środowiskową `PYTHONPATH`?
3. Wymień pięć głównych komponentów ścieżki wyszukiwania importowanych modułów.
4. Podaj cztery typy plików, jakie Python może załadować w odpowiedzi na operację importowania.
5. Czym jest przestrzeń nazw i co zawiera przestrzeń nazw modułu?

Sprawdź swoją wiedzę — odpowiedzi

1. Plik z kodem źródłowym automatycznie staje się obiektem modułu po zimportowaniu tego modułu. Z technicznego punktu widzenia kod źródłowy modułu jest wykonywany w czasie operacji importowania, po jednej instrukcji na raz, a wszystkie nazwy przypisane w tym czasie stają się atrybutami obiektu modułu.
2. Zmienną `PYTHONPATH` ustawiamy tylko po to, aby móc importować moduły z katalogów innych niż katalog roboczy (czyli katalog bieżący w czasie pracy w sesji interaktywnej lub katalog zawierający plik najwyższego poziomu). W praktyce w przypadku bardziej złożonych programów jest to powszechnie stosowane rozwiązanie.
3. Pięć podstawowych komponentów ścieżki wyszukiwania importowanych modułów to katalog główny skryptu najwyższego poziomu (katalog zawierający ten skrypt),

wszystkie katalogi wymienione w zmiennej środowiskowej PYTHONPATH, katalogi biblioteki standardowej, wszystkie katalogi wymienione w plikach ścieżek *.pth* umieszczonych w standardowych miejscach oraz podkatalog *site-packages*, w którym instalowane są rozszerzenia zewnętrzne. Z tego wszystkiego programiści mogą dostosowywać do własnych potrzeb zmienną PYTHONPATH oraz pliki *.pth*.

4. Python może załadować plik z kodem źródłowym (*.py*), kodem bajtowym (*.pyc* lub *.pyo*), moduł rozszerzenia w języku C (na przykład plik *.so* w systemie Linux lub *.dll* albo *.pyd* w systemie Windows) lub katalog o danej nazwie w przypadku importowania pakietów. Operacje importowania mogą również ładować bardziej egzotyczne komponenty, takie jak części archiwów ZIP, klasy języka Java w dystrybucji Jython, komponenty platformy .NET w dystrybucji IronPython, a także statycznie dołączone rozszerzenia w języku C, które nie mają żadnych plików. Dzięki punktom zaczepienia operacji importowania można ładować prawie wszystko.
 5. Przestrzeń nazw to samodzielny pakiet zmiennych znanych jako *atrybuty* obiektu przestrzeni nazw. Przestrzeń nazw modułu zawiera wszystkie nazwy przypisane na najwyższym poziomie pliku modułu (czyli niezagnieżdżone w instrukcjach *def* lub *class*). Z technicznego punktu widzenia zakres globalny modułu staje się przestrzenią nazw atrybutów obiektu modułu. Przestrzeń nazw modułu można również modyfikować za pomocą operacji przypisania w plikach innych od importujących moduł, niemniej wielu użytkownikom się to nie podoba (więcej informacji na ten temat znajdziesz w rozdziale 17.).
-

[1] Tak naprawdę podawanie ścieżki oraz rozszerzenia nazwy pliku w instrukcji *import* jest niepoprawne składniowo. Jednak *importowanie pakietów*, omówione w rozdziale 24., pozwala na umieszczenie w instrukcji *import* części ścieżki do katalogu prowadzącej do pliku w postaci zbioru nazw rozdzielonych kropkami. Jednak importowanie pakietów wykorzystuje normalną ścieżkę wyszukiwania modułów, aby zlokalizować katalog znajdujący się najbardziej z lewej strony ścieżki pakietu (jest zatem względne w stosunku do katalogu ze ścieżki wyszukiwania). W instrukcjach *import* nie można również używać składni specyficznych dla danej platformy; taka składnia działa jedynie w ścieżce wyszukiwania. Powinieneś również pamiętać, że problemy ze ścieżką wyszukiwania modułów nie są tak istotne, kiedy używamy *zamrożonych plików binarnych* (pisaliśmy o nich w rozdziale 2.), które osadzają kod bajtowy w obrazie binarnym.

[2] Jak wspominaliśmy wcześniej, Python przechowuje zaimportowane moduły we wbudowanym słowniku *sys.modules*, dzięki czemu może śledzić, które z nich zostały już załadowane. Jeżeli chcemy sprawdzić, które moduły zostały już załadowane, należy zaimportować moduł *sys* i wyświetlić *list(sys.modules.keys())*. Więcej szczegółowych informacji na temat tej wewnętrznej tabeli znajdziesz w rozdziale 25.

[3] Więcej szczegółowych informacji na temat nowej składni *importowania względnego* i reguły wyszukiwania w Pythonie 3.x znajdziesz w rozdziale 24.; importowanie względne modyfikuje ścieżkę wyszukiwania dla instrukcji *from*, kiedy użyte zostaną znaki kropki (na przykład *from . import string*). Domyślnie w Pythonie 3.x własny katalog pakietu nie jest przeszukiwany, o ile importowanie względne nie jest wykorzystywane przez pliki w samym pakiecie.

Rozdział 23. Podstawy tworzenia modułów

Skoro już przyjrzaliśmy się podstawowym koncepcjom stojącym za modułami, czas przejść do prostych przykładów modułów w działaniu. Chociaż niektóre z omawianych na początku zagadnień mogą uważnym czytelnikom wydawać się pewnym powtórzeniem tego, o czym pisaliśmy w przykładach z poprzednich rozdziałów, szybko rozszerzymy je o dalsze szczegóły dotyczące modułów Pythona, o których jeszcze nie wspominaliśmy, takich jak zagnieżdżanie, przeładowywania, zasięgi i inne.

Moduły Pythona łatwo jest *utworzyć* — są to po prostu pliki z kodem w języku Python utworzone za pomocą edytora tekstowego. Nie musimy używać żadnych specjalnych poleceń, aby przekazać Pythonowi, że tworzymy moduł — wystarczy niemal dowolny plik tekstowy. Ponieważ Python sam radzi sobie ze szczegółami odnajdywania i ładowania modułów, są one również bardzo proste w *użyciu*. Klient po prostu importuje moduł lub poszczególne nazwy przez niego definiowane i wykorzystuje obiekty, do których się te nazwy odnoszą.

Tworzenie modułów

Aby zdefiniować moduł, wystarczy wykorzystać posiadany edytor tekstu do wpisania kodu Pythona do pliku tekstowego i zapisać ten plik z rozszerzeniem `.py`. Każdy taki plik jest automatycznie uznawany za moduł Pythona. Wszystkie nazwy przypisane na najwyższym poziomie modułu stają się jego *atrybutami* (nazwami powiązanymi z obiektem modułu) i są eksportowane do klientów w celu użycia — zmieniają się automatycznie ze zmiennych na atrybuty obiektów modułu.

Jeżeli na przykład wpiszemy poniższą instrukcję `def` do pliku o nazwie `module1.py`, a następnie zaimportujemy ten plik, utworzymy obiekt modułu z jednym atrybutem — zmienną `printer`, która okazuje się referencją do obiektu funkcji.

```
def printer(x):                                # Atrybut modułu
    print(x)
```

Nazwy modułów

Zanim przejdziemy dalej, powiniensem powiedzieć kilka słów na temat nazw plików modułów. Moduły możemy nazywać w dowolny sposób, jednak jeżeli planujemy je w przyszłości importować, ich pliki powinny mieć rozszerzenie `.py`. Rozszerzenie to jest opcjonalne dla plików najwyższego poziomu, które będziemy wykonywać, ale nie importować — dodanie `.py` w każdym przypadku sprawia, że typ pliku będzie bardziej oczywisty, i w przyszłości pozwoli zaimportować wszystkie pliki.

Ponieważ nazwy modułów (bez rozszerzenia `.py`) stają się wewnątrz programu Pythona nazwami zmiennymi, powinny być zgodne z normalnymi regułami dotyczącymi nazw zmiennych, przedstawionymi w rozdziale 11. Możemy na przykład utworzyć moduł o nazwie

if.py, jednak nie możemy go zaimportować, ponieważ `if` jest słowem zarezerwowanym — kiedy spróbujemy wykonać instrukcję `import if`, otrzymamy błąd składni. Tak naprawdę zarówno nazwy plików modułów, jak i nazwy katalogów wykorzystywanych w importowaniu pakietów (omówionych w kolejnym rozdziale) muszą być zgodne z regułami dotyczącymi nazw zmiennych zaprezentowanymi w rozdziale 11. Mogą one w związku z tym zawierać wyłącznie litery, cyfry oraz znaki `_`. Nazwy katalogów pakietów nie mogą również zawierać żadnej składni specyficznej dla określonej platformy, w tym na przykład spacji.

Kiedy moduł jest importowany, Python odwzorowuje wewnętrzną nazwę modułu na zewnętrzną nazwę pliku, dodając ścieżkę do katalogu ze ścieżki wyszukiwania modułów na początku, a rozszerzenie `.py` (lub inne) na końcu. Moduł o nazwie `M` może na przykład zostać odwzorowany na jakiś plik zewnętrzny `<katalog>|M.<rozszerzenie>` zawierający kod modułu.

Inne rodzaje modułów

Jak wspominaliśmy w poprzednim rozdziale, można również utworzyć moduł Pythona, pisząc kod w języku takim jak C czy C++ czy innych (lub w przypadku implementacji Jython — również w Javie). Takie moduły znane są jako *moduły rozszerzeń* i są najczęściej używane do opakowania zewnętrznych bibliotek do użycia w skryptach Pythona. Po zimportowaniu przez kod Pythona moduły rozszerzeń wyglądają i działają tak samo jak moduły utworzone jako pliki z kodem źródłowym Pythona — dostęp do nich odbywa się za pomocą instrukcji `import` i udostępniają one funkcje oraz obiekty jako atrybuty modułu. Moduły rozszerzeń wykraczają poza zakres niniejszej książki; więcej informacji na ich temat można znaleźć w dokumentacji Pythona lub bardziej zaawansowanych książkach, takich jak *Programming Python*.

Używanie modułów

Klient może wykorzystywać napisane przez nas proste pliki modułów, używając do tego instrukcji `import` lub `from`. Obie instrukcje odnajdują, komplikują oraz wykonują kod pliku modułu, jeśli nie został on jeszcze załadowany. Podstawowa różnica polega na tym, że instrukcja `import` pobiera moduł jako całość, zatem by pobrać poszczególne nazwy, musimy skorzystać ze składni kwalifikującej (z kropką). W przeciwnieństwie do tego instrukcja `from` pobiera (lub kopiuje) określone zmienne z modułu.

Zobaczmy, co to wszystko oznacza dla kodu. Wszystkie poniższe przykłady w rezultacie wywołują funkcję `printer` zdefiniowaną w przedstawionym wyżej pliku modułu `module1.py`, jednak robią to w różny sposób.

Instrukcja `import`

W pierwszym przykładzie nazwa `module1` spełnia dwa różne cele — identyfikuje plik zewnętrzny, który ma zostać załadowany, i staje się zmienną w skrypcie, będącą po załadowaniu pliku odniesieniem do obiektu modułu.

```
>>> import module1                                # Pobranie modułu jako całości  
(jednego lub więcej)  
>>> module1.printer('Witaj, świecie!')           # Zapis kwalifikowany w celu  
otrzymania nazwy  
Witaj, świecie!
```

W instrukcji `import` po prostu umieszczamy jedną lub więcej nazw modułów do załadowania, oddzielonych od siebie przecinkami. Ponieważ instrukcja `import` daje nazwę, która odnosi się

do całego obiektu modułu, odwołując się do atrybutów modułu, musimy podać jego nazwę (na przykład `module1.printer`).

Instrukcja `from`

W przeciwieństwie do `import` instrukcja `from` kopiuje *wybrane nazwy* z danego pliku do innego zasięgu, dlatego pozwala na bezpośrednie używanie skopiowanych nazw w skrypcie, bez dodawania do nich nazwy modułu (wystarczy zatem użyć samego `printer`):

```
>>> from module1 import printer      # Skopiowanie zmiennej (jednej lub
   więcej)
>>> printer('Witaj, świecie!')       # Nie ma konieczności użycia nazwy
modułu (nazwy kwalifikowanej)

Witaj, świecie!
```

Taka forma instrukcji `from` pozwala na użycie listy składającej się z jednej lub więcej nazw do skopiowania, oddzielonych od siebie przecinkami. W naszym przypadku ma to taki sam efekt jak w poprzednim przykładzie, jednak ponieważ importowana zmienna jest kopowana do zasięgu, w którym pojawia się instrukcja `from`, użycie tej zmiennej w skrypcie wymaga nieco mniejszej ilości pisania — nazwy zmiennej możemy użyć bezpośrednio, bez podawania jednocześnie nazwy zawierającego ją modułu. W praktyce musimy tak zrobić, ponieważ instrukcja `from` nie przypisuje automatycznie nazwy modułu.

Jak zobaczymy w szczegółach później, instrukcja `from` jest jedynie nieznacznym rozszerzeniem instrukcji `import` — importuje ona plik modułu tak jak zawsze (wykonując pełną, trójetapową procedurę importowania, o której pisaliśmy w poprzednim rozdziale), ale dodaje dodatkowy krok kopiący jedną lub więcej nazw (a nie obiektów) z tego pliku. Cały plik modułu jest ładowany, ale do dyspozycji masz tylko nazwy umożliwiające bardziej bezpośredni dostęp do jego komponentów.

Instrukcja `from *`

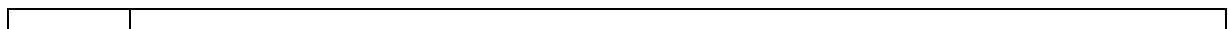
Kolejny przykład wykorzystuje specjalną formę instrukcji `from`. Kiedy użyjemy znaku `*`, otrzymujemy kopie *wszystkich* zmiennych przypisanych na najwyższym poziomie modułu. Tutaj możemy znowu wykorzystać nazwę `printer` w skrypcie bez dodawania do niej nazwy modułu.

```
>>> from module1 import *          # Skopiowanie wszystkich
   zmiennych
>>> printer('Witaj, świecie!')

Witaj, świecie!
```

Z technicznego punktu widzenia instrukcje `import` oraz `from` wywołują tę samą operację importowania. Forma `from *` po prostu dodaje jeszcze jeden krok kopiący wszystkie zmienne z modułu do zakresu importującego. W rezultacie przestrzeń nazw jednego modułu zostaje tak naprawdę włączona do przestrzeni nazw drugiego. Oznacza to dla nas mniej pisania. Pamiętaj, że gwiazdka (*) działa tylko w tym kontekście; nie możesz używać wzorca dopasowania do wybierania podzbioru nazw z modułu (choć możesz utworzyć pętlę przechodzącą przez atrybut `__dict__` modułu, co wymaga jednak nieco większej ilości kodu; opowiem o tym już za chwilę).

I to tyle — moduły są naprawdę proste w użyciu. Aby jednak lepiej zrozumieć, co tak naprawdę dzieje się, kiedy definiujemy i wykorzystujemy moduły, przyjrzyjmy się bardziej szczegółowo niektórym ich właściwościom.





W Pythonie 3.x opisana tutaj postać `from ... *` może być zastosowana **jedynie** na najwyższym poziomie pliku modułu, a nie wewnątrz funkcji. Python 2.x pozwala na użycie jej wewnątrz funkcji, jednak wyświetla odpowiednie ostrzeżenie. W praktyce użycie tej instrukcji wewnątrz funkcji spotyka się wyjątkowo rzadko. Kiedy tak się dzieje, Python nie jest w stanie wykrywać zmiennych statycznie, przed wykonaniem funkcji. Najlepsze praktyki programowania we wszystkich wersjach Pythona zalecają umieszczanie *wszystkich* instrukcji importowania na początku pliku modułu; nie jest to co prawda wymagane składowo, ale ułatwia ich zauważenie.

Operacja importowania jest przeprowadzana tylko raz

Jedno z pytań, najczęściej zadawanych przez początkujących użytkowników próbujących korzystać z modułów, brzmi: „Dlaczego moje importy przestały działać?”. Użytkownicy często zgłoszą, że pierwsza operacja importowania działa dobrze, natomiast późniejsze w czasie sesji interaktywnej (lub wykonywania programu) wydają się nie przynosić efektu. Wszystko jest jednak w porządku — tak naprawdę kolejne operacje importu wcale nie powinny przynosić efektów i zaraz wyjaśnimy, dlaczego tak się dzieje.

Moduły są ładowane i wykonywane tylko przy pierwszej instrukcji `import` lub `from`. Jest to celowe — ponieważ importowanie jest kosztowną operacją, Python domyślnie wykonuje ją tylko raz na plik i raz na proces. Późniejsze operacje importowania po prostu pobierają już załadowany obiekt modułu.

Kod inicjalizujący

W konsekwencji, ponieważ kod najwyższego poziomu w pliku modułu jest zazwyczaj wykonywany tylko raz, możemy go wykorzystać do inicjalizacji zmiennych. Rozważmy na przykład plik `simple.py`.

```
print('witam')
spam = 1                                # Inicjalizacja zmiennej
```

W powyższym przykładzie instrukcje `print` oraz `=` wykonywane są za pierwszym razem, gdy moduł jest importowany, natomiast zmienna `spam` inicjalizowana jest w czasie importowania.

```
% python
>>> import simple                      # Pierwszy import: ładuje i wykonuje
      kod pliku
      witam
      >>> simple.spam                      # Przypisanie tworzy atrybut
      1
```

Druga i kolejne operacje importowania nie wykonują ponownie kodu modułu, a po prostu pobierają już utworzony obiekt modułu z wewnętrznej tabeli modułów Pythona. Z tego powodu zmienna `spam` nie zostanie zainicjalizowana ponownie.

```
>>> simple.spam = 2                      # Modyfikacja atrybutu w module
      >>> import simple                      # Pobiera już załadowany moduł
      >>> simple.spam                      # Kod nie został wykonany ponownie -
      atrybut nie zmienia się
```

Oczywiście czasami naprawdę *chcemy*, aby kod modułu był wykonany ponownie w kolejnej operacji importowania. W dalszej części rozdziału zobaczymy, jak można to uzyskać za pomocą funkcji Pythona `reload`.

Instrukcje import oraz from są przypisaniami

Tak jak `def`, instrukcje `import` oraz `from` są *instrukcjami wykonywalnymi*, a nie deklaracjami w czasie komplikacji. Mogą one być między innymi zagnieżdżane w testach `if`, mogą pojawiać się w instrukcjach `def` funkcji (wtedy odpowiednie moduły będą ładowane tylko po wywołaniu funkcji), mogą też pojawiać się w instrukcjach `try`, zapewniając wartości domyślne i tak dalej. Nie są wykonywane, dopóki Python nie dojdzie do nich w kodzie w czasie wykonywania programu. Innymi słowy, importowane moduły oraz zmienne nie są dostępne, dopóki nie zostaną wykonane powiązane z nimi instrukcje `import` lub `from`.

Modyfikowanie elementów mutowalnych w modułach

Podobnie jak def, import oraz from są niejawnymi przypisami.

- Instrukcja `import` przypisuje cały obiekt modułu do jednej nazwy.
 - Instrukcja `from` przypisuje jedną lub więcej zmiennych do obiektów o tych samych nazwach w innym module.

Wszystkie omówione dotychczas kwestie dotyczące przypisania dotyczą również dostępu do modułu. Na przykład zmienna skopiowana za pomocą instrukcji `from` staje się referencją do współdzielonego obiektu. Tak jak w przypadku argumentów funkcji, ponowne przypisanie zmiennej nie ma wpływu na moduł, z którego została ona skopiowana, jednak modyfikacja pobranego *obiektu mutowalnego* może spowodować jego zmianę w module, z którego został on zimportowany. Aby to zilustrować, rozważmy poniższy plik `small.py`.

$$\begin{aligned}x &= 1 \\y &= [1, 2]\end{aligned}$$

Podczas importowania za pomocą `from` kopujemy nazwy do zasięgu importera początkowo udostępniającego obiekty, do których odwołują się nazwy modułu:

```
% python
>>> from small import x, y          # Skopiowanie dwóch zmiennych z
modułu
>>> x = 42                         # Modyfikacja jedynie lokalnej
zmiennej x
>>> y[0] = 42                       # Modyfikacja współdzielonego
objektu mutowalnego w miejscu
```

W powyższym kodzie `x` nie jest współdzielonym obiektem mutowalnym, natomiast `y` nim jest. Zmienna `y` w kodzie importującym i w kodzie importowanym odnosi się do tego samego obiektu listy, dlatego jej modyfikacja w jednym miejscu powoduje zmianę w innym.

```
>>> import small # Pobranie nazwy modułu (from tego nie robi)
>>> small.x # x z modułu small nie jest moim x
1
>>> small.y # Współdzielimy jednak zmienny obiekt
```

[42, 2]

Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 6. Aby lepiej sobie wyobrazić, co przypisania `from` robią z referencjami, warto wrócić do rysunku 18.1 przedstawiającego przekazywanie argumentów funkcji i zastąpić w nim „wywołującego” oraz „funkcję” odpowiednio „importowanym” i „importującym”. Rezultat będzie taki sam, tyle że tutaj mamy do czynienia ze zmiennymi w modułach, a nie w funkcjach. Przypisanie działa w Pythonie tak samo w każdych okolicznościach.

Modyfikowanie nazw pomiędzy plikami

W poprzednim przykładzie widać było, że przypisanie do `x` w sesji interaktywnej modyfikowało zmienną `x` tylko w tym zakresie, a nie miało wpływu na zmienną `x` z pliku. Nie istnieje żadne połączenie pomiędzy zmienną skopiowaną za pomocą instrukcji `from` a plikiem, z którego zmienna ta pochodzi. Aby naprawdę zmodyfikować zmienną globalną z innego pliku, musimy użyć instrukcji `import`.

```
% python
>>> from small import x, y          # Skopiowanie dwóch zmiennych z
modułu
>>> x = 42                          # Modyfikacja jedynie lokalnej
zmiennej x
>>> import small                   # Pobranie nazwy modułu
>>> small.x = 42                   # Modyfikuje zmienną x w innym
module
```

To zjawisko zostało objaśnione w rozdziale 17. Ponieważ modyfikacja zmiennych w innych modułach jest znanym źródłem nieporozumień (a często również złym wyborem projektowym), powrócimy do tej techniki w dalszej części książki. Warto zauważyć, że zmiana `y[0]` w poprzedniej sesji jest inna — modyfikuje ona *obiekt*, a nie nazwę zmiennej, zatem nazwy w obu modułach nadal odwołują się do tego samego, zmodyfikowanego obiektu.

Równoważność instrukcji `import` oraz `from`

Warto zwrócić uwagę na to, że w poprzednim przykładzie musielismy wykonać instrukcję `import` po instrukcji `from` w celu uzyskania dostępu do nazwy modułu `small`. Instrukcja `from` kopiuje jedynie nazwy z jednego modułu do drugiego i nie przypisuje samej nazwy modułu. Przynajmniej z koncepcjonalnego punktu widzenia poniższa instrukcja `from`:

```
from module import name1, name2      # Skopiowanie tylko tych dwóch
nazw z modułu
```

jest odpowiednikiem poniższej sekwencji instrukcji:

```
import module                         # Pobranie obiektu modułu
name1 = module.name1                  # Skopiowanie zmiennych przez
przypisanie
name2 = module.name2
del module                            # Pozbycie się nazwy modułu
```

Jak wszystkie przypisania, instrukcje `from` tworzą w kodzie importującym nowe zmienne, które początkowo odnoszą się do obiektów o tych samych nazwach w pliku importowanym. Kopowane są jednak jedynie *nazwy*, a nie obiekty, do których się odwołują, ani nie sama nazwa modułu. Kiedy skorzystamy z formy `from *` (`from module import *`), równoważność instrukcji

pozostaje bez zmian, jednak w ten sposób do zakresu importującego kopowane są wszystkie zmienne najwyższego poziomu.

Warto zwrócić uwagę na to, że pierwszy krok instrukcji `from` wykonuje normalną operację `import`, z całą semantyką tej operacji, opisaną w poprzednim rozdziale. Z tego powodu `from` zawsze importuje *cały* moduł do pamięci (jeżeli jeszcze nie został on zimportowany), bez względu na to, ile zmiennych kopujemy z pliku. Nie da się załadować jedynie części pliku modułu (na przykład jednej funkcji), jednak ponieważ moduły są w Pythonie kodem bajtowym, a nie maszynowym, wpływ takiego rozwiązania na wydajność jest na ogół znikomy.

Potencjalne pułapki związane z użyciem instrukcji `from`

Ponieważ instrukcja `from` sprawia, że lokalizacja zmiennych staje się mniej jawną i zagmatwana (zmienna `name` jest mniej zrozumiała dla osoby analizującej kod od zmiennej `module.name`), niektórzy użytkownicy Pythona zalecają jak najczęściej zamiast `from` używać instrukcji `import`. Nie jestem jednak pewien, czy taka rada jest uzasadniona — instrukcja `from` jest powszechnie używana bez jakichś poważniejszych negatywnych konsekwencji. W praktyce, w prawdziwych programach, często wygodniej jest nie wpisywać pełnej nazwy modułu za każdym razem, kiedy chcemy skorzystać z jego narzędzi. Jest to szczególnie prawdziwe w przypadku dużych modułów udostępniających wiele atrybutów — na przykład modułu graficznego interfejsu użytkownika `tkinter` z biblioteki standardowej.

To prawda, że instrukcja `from` może potencjalnie powodować uszkodzenia w przestrzeniach nazw, przynajmniej teoretycznie — jeżeli będziemy ją wykorzystywać do importowania zmiennych, które mają takie same nazwy jak istniejące zmienne z naszego zasięgu, takie zmienne zostaną po cichu nadpisane. Ten problem nie występuje w przypadku prostych instrukcji `import`, ponieważ w ich przypadku zawsze trzeba przejść nazwę modułu w celu dotarcia do jego zawartości (`module.attr` nie będzie w konflikcie ze zmienną o nazwie `attr` w naszym zasięgu). Dopóki rozumiemy ten sposób działania instrukcji `from` i jesteśmy na niego przygotowani, nie jest to w praktyce wielkim problemem, w szczególności kiedy w jasny sposób wymienimy importowane zmienne (na przykład `from module import x, y, z`).

Z drugiej strony, instrukcja `from` powoduje poważniejsze problemy, kiedy używa się jej w połączeniu z wywołaniem funkcji `reload`, gdyż importowane nazwy mogą odnosić się do wcześniejszych wersji obiektów. Co więcej, forma `from module import *` naprawdę może spowodować uszkodzenia w przestrzeniach nazw i sprawić, że nazwy będą trudne do zrozumienia — zwłaszcza gdy zastosuje się ją do większej liczby plików. W takim przypadku nie da się powiedzieć, z którego modułu pochodzi dana nazwa — poza, oczywiście, przeszukaniem zewnętrznych plików źródłowych. W rezultacie forma ze znakiem `*` wyłącza jedną przestrzeń nazw do drugiej i tym samym skutecznie neutralizuje ogromną zaletę modułów, jaką jest separacja przestrzeni nazw. Kwestie te omówimy bardziej szczegółowo w podrozdziale „Pułapki związane z modułami” na końcu tej części książki (w rozdziale 25.).

Chyba najlepszą praktyczną radą, jakiej można udzielić w tym temacie, jest wybieranie `import` w przypadku prostych modułów, jawnie wymienianie importowanych zmiennych w większości instrukcji `from` i ograniczenie formy `from *` do jednej operacji importowania na plik. W ten sposób możemy zakładać, że wszystkie niezdefiniowane zmienne pochodzą z modułu, do którego odnosimy się za pomocą formy `from *`. Wykorzystywanie instrukcji `from` wymaga nieco uwagi, ale po użbrojeniu się w odrobinę wiedzy większość programistów dochodzi najczęściej do wniosku, że jest to wygodny sposób dostępu do modułów.

Kiedy wymagane jest stosowanie instrukcji `import`

Jedyna sytuacja, kiedy naprawdę *musisz* używać instrukcji `import` zamiast `from`, występuje wtedy, kiedy musisz użyć tej samej nazwy zdefiniowanej w dwóch różnych modułach. Na przykład kiedy ta sama nazwa jest różnie zdefiniowana w dwóch różnych plikach:

```

# Plik M.py
def func():
    ...coś robi...
# Plik N.py
def func():
    ...robi coś innego...

```

i musimy w programie wykorzystać obie wersje tej zmiennej, instrukcja `from` nie przyda się nam do niczego — w jednym zasięgu możemy mieć tylko jedno przypisanie do zmiennej.

```

# Plik 0.py
from M import func
from N import func                                # Nadpisuje nazwę otrzymaną z
modułu M
func()                                              # Wywołuje tylko N.func

```

W tym przypadku zadziała jednak instrukcja `import`, ponieważ uwzględnienie nazwy modułu sprawia, że zmienne stają się unikalne.

```

# Plik 0.py
import M, N                                      # Pobranie całych modułów, a nie
tylko wybranych nazw
M.func()                                            # Teraz możemy wywołać obie
nazwy
N.func()                                            # Dzięki nazwom modułów są one
unikalne

```

Taka sytuacja zdarza się na tyle rzadko, że w praktyce mało prawdopodobne jest, iż się z nią spotkamy. Jeżeli tak się jednak stanie, użycie instrukcji `import` pozwala zapobiec konfliktowi między nazwami zmiennych. Innym sposobem rozwiązania tego problemu jest użycie rozszerzenia `as`, które omówimy szczegółowo w rozdziale 25., ale które jest wystarczająco proste, abyśmy je mogli wprowadzić już tutaj:

```

# 0.py
from M import func as mfunc                      # Za pomocą as zmieniamy nazwę
na unikatową
from N import func as nfunc
mfunc(); nfunc()                                    # Wywołujemy jedną lub drugą w
miarę potrzeb

```

Rozszerzenie `as` działa w instrukcjach `import` i `from` jako proste narzędzie do zmiany nazw (można go również użyć do zdefiniowania krótszego synonimu długiej nazwy modułu podczas importu); więcej szczegółowych informacji na temat tego rozwiązania znajdziesz w rozdziale 25.

Przestrzenie nazw modułów

Moduły najlepiej jest sobie chyba wyobrazić jako pakiety nazw, czyli miejsca, w których definiujemy nazwy, jakie chcemy pokazać reszcie systemu. Z technicznego punktu widzenia moduły zazwyczaj odpowiadają plikom, a Python tworzy obiekt modułu zawierający wszystkie nazwy przypisane w pliku modułu. W uproszczeniu moduły są jedynie przestrzeniami nazw (miejscami, w których tworzone są nazwy), a nazwy istniejące w module nazywane są jego *atrybutami*. W niniejszym podrozdziale wyjaśnimy, jak to wszystko działa.

Pliki generują przestrzenie nazw

Jak już wspominaliśmy, pliki stają się przestrzeniami nazw, ale jak to się naprawdę dzieje? Mówiąc w uproszczeniu, każda nazwa, do której na najwyższym poziomie pliku (czyli poza zagnieżdżeniem w ciele funkcji lub klasy) przypisywana jest wartość, staje się atrybutem modułu.

Na przykład, kiedy mamy instrukcję przypisania `X = 1` na najwyższym poziomie pliku modułu `M.py`, zmienna `X` staje się atrybutem modułu `M`, do którego możemy się spoza modułu odnosić przez `M.x`. Zmienna `X` staje się również zmienną globalną dla pozostałego kodu wewnętrz modułu `M.py`, jednak żeby to zrozumieć, musimy wyjaśnić nieco bardziej szczegółowo pojęcie ładowania modułów oraz zasięgów.

- **Instrukcje modułów wykonywane są przy pierwszej operacji importowania.** Za pierwszym razem, gdy moduł jest importowany w dowolnym miejscu systemu, Python tworzy pusty obiekt modułu i wykonuje instrukcje z pliku modułu jedna po drugiej, od góry do dołu.
- **Przypisania na najwyższym poziomie pliku tworzą atrybuty modułów.** W czasie operacji importowania instrukcje przypisujące nazwy na najwyższym poziomie pliku (niezagnieżdżone w funkcji lub klasie), takie jak `def` czy `=`, tworzą atrybuty obiektu modułu. Przypisane nazwy przechowywane są w przestrzeni nazw modułu.
- **Dostęp do przestrzeni nazw modułu odbywa się za pomocą atrybutu `_dict_` lub `dir(M)`.** Przestrzeń nazw modułów utworzone przez operacje importowania są słownikami. Dostęp do nich można uzyskać za pomocą wbudowanego atrybutu `_dict_` powiązanego z obiektami modułów, a można go sprawdzić za pomocą funkcji `dir`. Funkcja `dir` jest przybliżonym odpowiednikiem listy posortowanych kluczy atrybutu `_dict_` obiektu, jednak zawiera odziedziczone zmienne klas, może nie być kompletna i może się zmieniać w kolejnych wydaniach Pythona.
- **Moduły są jednym zasięgiem (zasięg lokalny jest w nich globalny).** Jak pokazywaliśmy w rozdziale 17., zmienne znajdujące się na najwyższym poziomie pliku modułu oddają się tym samym regułom przypisania i referencji co zmienne funkcji, jednak zasięgi lokalny i globalny są tym samym (z formalnego punktu widzenia są zgodne z regułą zasięgów LEGB z rozdziału 17., jednak bez warstw wyszukiwania `L` oraz `E`).

Co najważniejsze, globalny *zasięg* modułu staje się słownikiem atrybutów *obiektu* modułu po załadowaniu tego modułu. W przeciwieństwie do zasięgów funkcji, gdzie zakres lokalny istnieje tylko na czas wykonywania funkcji, zasięg pliku modułu staje się przestrzenią nazw atrybutów obiektu modułu i *istnieje* także po zakończeniu operacji importowania, zapewniając dostęp do nazw i narzędzi dla kodu importującego.

Poniżej znajduje się demonstracja tych koncepcji. Założmy, że utworzymy następujący plik modułu w edytorze tekstu i nazwiemy go `module2.py`.

```
print('rozpoczęto ładowanie...')

import sys

name = 42

def func(): pass
```

```
class klass: pass  
print('zakończono ładowanie.')
```

Przy pierwszym importie modułu (lub wykonaniu go jako programu) Python wykonuje jego instrukcje od góry do dołu. Niektóre instrukcje tworzą zmienne w przestrzeni nazw modułu, inne mogą wykonywać jakieś zadania w czasie importowania modułu. Przykładowo dwie instrukcje print w tym pliku wykonywane są w czasie importowania.

```
>>> import module2  
rozpoczęto ładowanie...  
zakończono ładowanie.
```

Po załadowaniu modułu jego zakres staje się przestrzenią nazw atrybutów w obiekcie modułów otrzymywanym z instrukcji import. Dostęp do tych atrybutów uzyskuje się, poprzedzając ich nazwy nazwą zawierającą je modułu.

```
>>> module2.sys  
<module 'sys' (built-in)>  
>>> module2.name  
42  
>>> module2.func  
<function func at 0x000000000222E7B8>  
>>> module2.klass  
<class 'module2(klass)'>
```

W powyższym kodzie zmienne sys, name, func oraz klass zostały przypisane w czasie wykonywania instrukcji modułu, dlatego po zakończeniu importowania są atrybutami. Klasy omówione zostaną w szóstej części książki, jednak już teraz możemy zwrócić uwagę na atrybut sys. Instrukcje import naprawdę przypisują obiekty modułów do nazw, a każdy rodzaj przypisania na najwyższym poziomie pliku generuje atrybut modułu.

Słowniki przestrzeni nazw: `__dict__`

Wewnętrzne przestrzenie nazw modułu przechowywane są jako obiekty słowników. Są to normalne obiekty słowników ze wszystkimi zwykłymi metodami słowników. W razie potrzeby — na przykład aby napisać narzędzia, które wyświetlają ogólną zawartość modułu, tak jak będziemy robić w rozdziale 25. — możemy uzyskać dostęp do słownika przestrzeni nazw za pomocą atrybutu `__dict__` modułu. Kontynuując przykład z poprzedniej sekcji (pamiętaj, aby w Pythonie 3.x opakować go w wywołanie funkcji list — jest to obiekt widoku, a jego zawartość w innych wersjach może się nieco różnić od pokazanej):

```
>>> list(module2.__dict__.keys())  
['__loader__', 'func', 'klass', '__builtins__', '__doc__', '__file__',  
'__name__',  
'name', '__package__', 'sys', '__initializing__', '__cached__']
```

Nazwy przypisane w pliku modułu wewnętrznie stają się kluczami słownika, dlatego większość nazw z tej listy odzwierciedla przypisania na najwyższym poziomie pliku. Python dodaje jednak również pewne zmienne do przestrzeni modułów za nas; na przykład, `__file__` podaje na przykład nazwę pliku, z którego załadowany został moduł, a `__name__` jego nazwę widoczną dla

kodu importującego (bez rozszerzenia .py oraz ścieżki do katalogu). Aby zobaczyć tylko nazwy, które przypisuje Twój kod, odfiltruj nazwy z podwójnym podkreśleniem, tak jak to robiliśmy wcześniej podczas pracy z poleceniem dir w rozdziale 15. i podczas omawiania wbudowanych zasięgów w rozdziale 17.:

```
>>> list(name for name in module2.__dict__.keys() if not
name.startswith('__'))
['func', 'klass', 'name', 'sys']
>>> list(name for name in module2.__dict__ if not name.startswith('__'))
['func', 'sys', 'name', 'klass']
```

Tym razem zamiast za pomocą listy filtrujemy za pomocą *generatora* i możemy pominąć `.keys()`, ponieważ słowniki generują klucze automatycznie, choć niejawnie; końcowy efekt jest jednak taki sam. Podobne słowniki `__dict__` zobaczymy przy okazji omawiania obiektów opartych na *klasach* w części VI. W obu przypadkach pobieranie atrybutów jest podobne do indeksowania słownika, chociaż tylko ten pierwszy sposób powoduje dziedziczenie w klasach:

```
>>> module2.name, module2.__dict__['name']
(42, 42)
```

Kwalifikowanie nazw atrybutów

Skoro już zapoznaliśmy się bliżej z modułami, powinniśmy się bardziej formalnie przyjrzeć pojęciu *kwalifikacji* zmiennych. W Pythonie możemy uzyskać dostęp do atrybutów dowolnego obiektu za pomocą składni *kwalifikującej* (zapis z kropką, używany przy pobieraniu atrybutów) w postaci *obiekt.atrybut*.

Kwalifikacja jest tak naprawdę wyrażeniem zwracającym wartość przypisaną do nazwy atrybutu powiązanego z obiektem. Przykładowo wyrażenie `module2.sys` z poprzedniego przykładu pobiera wartość przypisaną do zmiennej `sys` z modułu `module2`. W podobny sposób, jeżeli mamy wbudowany obiekt listy `L`, wyrażenie `L.append` zwraca metodę `append` obiektu powiązanego z tą listą.

Zawsze powinieneś pamiętać, że kwalifikacja atrybutów nie ma nic wspólnego z regułami zasięgów, które badaliśmy w rozdziale 17.; jest to całkowicie niezależna koncepcja. Kiedy używamy kwalifikacji do uzyskania dostępu do nazw, podajemy Pythonowi obiekt, z którego ma pobrać określone atrybuty. Reguła LEGB ma zastosowanie jedynie do „gołych” nazw niekwalifikowanych — można jej użyć dla nazwy znajdującej się najbardziej na lewo w ścieżce nazw, ale kolejne nazwy po kropkach będą zamiast tego powodowały wyszukiwanie określonych obiektów. Reguły są zatem następujące:

Proste zmienne

`X` oznacza wyszukanie zmiennej `X` w zasięgu bieżącym (zgodnie z regułą LEGB, którą opisywaliśmy w rozdziale 17.).

Kwalifikacja

`X.Y` oznacza wyszukiwanie `X` w zasięgach bieżących, a następnie poszukiwanie atrybutu `Y` w obiekcie `X` (a nie w zasięgach).

Ścieżki kwalifikacji

`X.Y.Z` oznacza wyszukiwanie zmiennej `Y` w obiekcie `X`, a następnie poszukiwanie zmiennej `Z` w obiekcie `X.Y`.

Uniwersalność

Kwalifikacja działa na wszystkich obiektach z atrybutami — modułach, klasach, typach będących rozszerzeniami w języku C itd.

W szóstej części książki zobaczymy, że kwalifikacja oznacza nieco więcej w przypadku klas (jest również miejscem, w którym zachodzi *dziedziczenie*), jednak ogólnie powyższe reguły odnoszą się do wszystkich nazw w Pythonie.

Importowanie a zasięgi

Jak już wiemy, nie da się uzyskać dostępu do nazw zdefiniowanych w innym pliku modułu bez uprzedniego zimportowania tego pliku. Oznacza to, że nigdy automatycznie nie zobaczymy nazw z innego pliku bez względu na strukturę importów lub wywołań funkcji w naszym programie. Znaczenie zmiennej jest zawsze uzależnione od lokalizacji przypisań w naszym kodzie źródłowym, a atrybutów obiektu zawsze żądamy w sposób jawnny.

Przykładowo rozważmy dwa poniższe proste moduły. Pierwszy (*moda.py*) definiuje zmienną X globalną dla kodu jego pliku wraz z funkcją modyfikującą tę zmienną globalną w następującym pliku:

```
X = 88                                     # X – zmienna globalna dla tego
pliku

def f():
    global X                                # Modyfikuje zmienną X tego
pliku

    X = 99                                  # Nie widzi nazw z innych
modułów
```

Drugi moduł (*modb.py*) definiuje własną zmienną globalną X, a potem importuje i wywołuje funkcję z pierwszego modułu.

```
X = 11                                     # X – zmienna globalna dla tego
pliku

import moda                                 # Dostęp do zmiennych z modułu
moda

moda.f()                                    # Ustawia zmienną moda.x, a nie
zmienną X z tego pliku

print(X, moda.x)
```

Po wykonaniu funkcja `moda.f` modyfikuje zmienną X z modułu `moda`, a nie zmienną X z modułu `modb`. Globalnym zasięgiem funkcji `moda.f` jest zawsze plik zawierający tę funkcję, bez względu na to, z którego modułu funkcja ta zostanie wywołana.

```
% python modb.py
11 99
```

Innymi słowy, operacje importowania nigdy nie przyznają kodowi z importowanego pliku widoczności „w góre” — importowany plik nie widzi zmiennych z pliku importującego. Bardziej formalnie:

- Funkcje nigdy nie widzą zmiennych innych funkcji, o ile nie są przez nie fizycznie otaczane.
- Kod modułu nigdy nie widzi zmiennych z innych modułów, o ile nie zostaną one zimportowane w sposób jawnny.

Takie zachowanie jest częścią koncepcji *zasięgów leksykalnych*. W Pythonie zasięgi otaczające fragment kodu są całkowicie uzależnione od fizycznej pozycji kodu w pliku. Wywołania funkcji czy operacje importowania modułów nigdy nie mają wpływu na zasięgi^[1].

Zagnieżdżanie przestrzeni nazw

W pewnym sensie, choć importowanie nie zagnieżdża przestrzeni nazw w góre, zagnieżdża je w dół. Oznacza to, że chociaż zimportowany moduł nigdy nie ma bezpośredniego dostępu do nazw w pliku, który go importuje, za pomocą ścieżek kwalifikacji atrybutów może zejść do dowolnie zagnieżdżonych modułów i uzyskać dostęp do ich atrybutów. Rozważmy na przykład kolejne trzy pliki. Plik *mod3.py* definiuje nazwę globalną oraz atrybut przez przypisanie.

```
X = 3
```

Plik *mod2.py* definiuje z kolei własną zmienną *X*, a następnie importuje *mod3* i wykorzystuje składnię kwalifikującą w celu uzyskania dostępu do atrybutu zimportowanego modułu.

```
X = 2

import mod3

print(X, end=' ')
# Globalna zmienna X pliku

print(mod3.x)
# Zmienna X z modułu mod3
```

Plik *mod1.py* również definiuje własną zmienną *X*, a następnie importuje *mod2* i pobiera atrybuty obu plików — *mod3.py* oraz *mod2.py*.

```
X = 1

import mod2

print(X, end=' ')
# Globalna zmienna X pliku

print(mod2.x, end=' ')
# Zmienna X z modułu mod2

print(mod2.mod3.x)
# Zmienna X z zagnieżdzonego
modułu mod3
```

Tak naprawdę, kiedy *mod1* importuje *mod2*, ustawia dwupoziomowe zagnieżdżenie przestrzeni nazw. Wykorzystując ścieżkę nazw *mod2.mod3.x*, może zejść aż do modułu *mod3* zagnieżdzonego w zimportowanym module *mod2*. Rezultat będzie taki, że *mod1* może widzieć zmienne *X* ze wszystkich trzech plików i tym samym ma dostęp do wszystkich trzech zasięgów globalnych.

```
% python mod1.py
```

```
2 3
```

```
1 2 3
```

Odwrotna zależność nie jest jednak prawdziwa — *mod3* nie widzi zmiennych z *mod2*, a *mod2* nie widzi tych z *mod1*. Przykład ten łatwiej jest zrozumieć, kiedy nie będziemy myśleć w kategoriach przestrzeni nazw i zasięgów, a zamiast tego skupimy się na obiektach. Wewnątrz modułu *mod1*, *mod2* jest po prostu zmienną odnoszącą się do obiektu z atrybutami, z których niektóre mogą się odnosić do innych obiektów z atrybutami (instrukcja *import* jest przypisaniem). W przypadku ścieżek takich, jak *mod2.mod3.x* Python po prostu oblicza je od lewej strony do prawej, po drodze pobierając atrybuty z obiektów.

Warto zauważyć, że kod modułu *mod1* może zawierać instrukcję *import mod2*, a następnie wyrażenie *mod2.mod3.x*, jednak nie może użyć instrukcji *import mod2.mod3* — taka składnia wywołuje tak zwane importowanie *pakietów* (katalogów) opisane w kolejnym rozdziale. Importowanie pakietów tworzy również zagnieżdżenie przestrzeni nazw modułu, jednak takie

instrukcje `import` służą do odzwierciedlania drzew katalogów, a nie prostych łańcuchów importu.

Przeładowywanie modułów

Jak widzieliśmy, kod modułu wykonywany jest domyślnie tylko raz na proces. Aby wymusić przeładowanie kodu modułu i ponowne jego wykonanie, trzeba zażądać tego od Pythona w sposób jawnym, wywołując jego wbudowaną funkcję `reload`. W niniejszym podrozdziale sprawdzimy, w jaki sposób można wykorzystać przeładowywane moduły i tym samym sprawić, że nasze systemy będą bardziej dynamiczne. W skrócie:

- Operacje importowania (wykonywane za pośrednictwem instrukcji `import` oraz `from`) ładują i wykonują kod modułu jedynie za pierwszym razem, gdy moduł jest importowany w danym procesie.
- Późniejsze operacje importowania wykorzystują załadowany już obiekt modułu bez przeładowywania go lub ponownego wykonywania kodu pliku.
- Funkcja `reload` wymusza ponowne załadowanie i wykonanie kodu załadowanego wcześniej modułu. Przypisania w nowym kodzie pliku modyfikują istniejący obiekt modułu w miejscu.

Skąd to całe zamieszanie związane z przeładowywaniem modułów? „Winę” za to ponosi mechanizm tzw. *dynamicznego dostosowywania*: funkcja `reload` pozwala na modyfikację części programu bez konieczności zatrzymywania całego programu. Dzięki zastosowaniu tej funkcji efekty zmian w komponentach mogą być zauważone natychmiast. Przeładowywane moduły nie pomaga w każdej sytuacji, ale tam, gdzie tak jest, bardzo skraca cykl tworzenia oprogramowania. Wyobraźmy sobie na przykład program bazodanowy, który na początku musi się połączyć z serwerem. Ponieważ modyfikacje programu czy jego dostosowanie do potrzeb użytkownika mogą być testowane natychmiast po przeładowaniu, wystarczy połączyć się tylko raz w czasie debugowania; w podobny sposób można aktualniać również dugo działające serwery.

Ponieważ Python jest językiem (mniej więcej) interpretowanym, pozbywa się kroków kompilowania i konsolidacji, które musimy przejść, by zmusić do działania program napisany w języku C. Moduły są ładowane dynamicznie, kiedy importowane są przez działający program. Przeładowywane oferuje dalszą przewagę w zakresie wydajności, pozwalając nam modyfikować części działających programów bez zatrzymywania ich.

Choć wykracza to już daleko poza zakres tej książki, warto jednak zauważyć, że przeładowanie przy użyciu funkcji `reload` obecnie działa tylko na modułach napisanych w języku Python; skompilowane moduły rozszerzeń zakodowane w języku takim jak C również można dynamicznie ładować w czasie działania programu, ale nie można ich przeładowywać (choć większość użytkowników woli jednak dostosowywać swój kod w Pythonie!).



Uwagi na temat wersji: W Pythonie 2.x funkcja `reload` dostępna jest w postaci funkcji wbudowanej. W Pythonie 3.x została jednak przeniesiona do modułu `imp` biblioteki standardowej — stąd w wersji 3.x występuje teraz jako `imp.reload`. Oznacza to po prostu, że w celu załadowania tego narzędzia (tylko w wersji 3.x) wymagane jest użycie dodatkowej instrukcji `import` lub `from`. Czytelnicy książki korzystający z wersji 2.x mogą zignorować te polecenia w przykładach lub po prostu je wykonać — w tej wersji funkcję `reload` można także znaleźć w module `imp`, co ma na celu ułatwienie przejścia na Pythona 3.x. Przeładowywane działa tak samo bez względu na to, w jakim pakiecie znajduje się funkcja `reload`.

Podstawy przeładowywania modułów

W przeciwnieństwie do instrukcji `import` oraz `from`:

- `reload` jest w Pythonie funkcją, a nie instrukcją;
- do `reload` przekazuje się istniejący obiekt modułu, a nie nową nazwę;
- funkcja `reload` w Pythonie 3.x znajduje się w module i przed użyciem sama musi zostać najpierw zimportowana.

Ponieważ funkcja `reload` oczekuje przekazania obiektu, przed przeładowaniem modułu trzeba go najpierw raz zimportować; jeżeli operacja importowania nie powiodła się z powodu jakiegoś błędu, będziesz musiał ją powtórzyć przed przeładowaniem modułu. Co więcej, składnie instrukcji `import` i wywołań funkcji `reload` różnią się od siebie: przeładowywanie wymaga zastosowania nawiasów, natomiast importowanie już nie. W dużym uproszczeniu przeładowywane wygląda mniej więcej tak:

```
import module                                # Początkowa operacja
importowania

...użycie składni moduł.atrybuty...

...
...                                         # Teraz modyfikacja pliku modułu

from imp import reload                         # Zimportowanie funkcji reload
(w wersji 3.x)

reload(module)                                 # Przeładowanie zimportowanego
modułu

...użycie składni moduł.atrybuty...
```

Typowy wzorzec użycia polega na zimportowaniu modułu, następnie modyfikacji jego kodu źródłowego w edytorze tekstu, a później przeładowaniu. Możemy tak postępować podczas pracy w sesji interaktywnej, ale także w większych programach, które ładują się okresowo.

Kiedy wywołamy funkcję `reload`, Python ponownie wczytuje kod źródłowy pliku modułu i raz jeszcze wykonuje jego instrukcje najwyższego poziomu. Chyba najważniejsze, o czym należy pamiętać w przypadku funkcji `reload`, jest to, że przeładowywany obiekt modułu jest modyfikowany w *miejscu* — funkcja nie usuwa obiektu modułu i nie tworzy go na nowo. Z tego powodu każde odwołanie do całego *obiektu* modułu w dowolnym miejscu programu jest automatycznie zależne od przeładowania. Wygląda to następująco:

- **Funkcja `reload` wykonuje kod przeładowywanej funkcji w bieżącej przestrzeni nazw tego modułu.** Ponowne wykonanie kodu modułu nadpisuje jego istniejącą przestrzeń nazw, zamiast ją usuwać i tworzyć na nowo.
- **Przypisania najwyższego poziomu w pliku następują nazwy nowymi wartościami.** Ponowne wykonanie instrukcji `def` zastępuje na przykład poprzednią wersję funkcji w przestrzeni nazw modułu, przypisując raz jeszcze nazwę tej funkcji.
- **Przeładowanie ma wpływ na wszystkie klienty wykorzystujące instrukcję `import` do pobrania modułów.** Ponieważ klient wykorzystujący instrukcję `import` używa składni kwalifikującej do pobierania atrybutów, po przeładowaniu modułu bez problemu odnajdzie nowe, zaktualizowane wartości.
- **Przeładowanie ma wpływ jedynie na przeszłe wywołanie instrukcji `from`.** Przeładowanie nie ma wpływu na klientów, którzy używali instrukcji `from` do pobierania atrybutów w przeszłości; nadal będą mieć odniesienia do starych obiektów pobranych przed przeładowaniem.

Przykład przeładowywania z użyciem reload

W celu zademonstrowania omówionych zagadnień poniżej zamieszczamy bardziej konkretny przykład zastosowania funkcji `reload`. W tym przykładzie zmodyfikujemy i przeładujemy plik modułu bez zatrzymywania sesji interaktywnej Pythona. Przeładowywane można również wykorzystywać w wielu innych sytuacjach (zobacz ramkę „Warto pamiętać: przeładowywanie modułów”), jednak dla ułatwienia wszystko tutaj nieco uprościmy. Najpierw w wybranym edytorze tekstu utworzymy plik modułu o nazwie `changer.py` i następującej zawartości.

```
message = "Pierwsza wersja"

def printer():
    print(message)
```

Moduł ten tworzy i eksportuje dwie nazwy — jedną powiązaną z łańcuchem znaków, drugą z funkcją. Teraz powinieneś uruchomić interpreter Pythona, zimportować moduł i wywołać eksportowaną przez niego funkcję. Funkcja wyświetla wartość zmiennej globalnej `message`.

```
% python
```

```
>>> import changer
>>> changer.printer()
```

```
Pierwsza wersja
```

Nie wyłączając interpretera, w osobnym oknie wprowadź zmiany w zimportowanym wcześniej pliku modułu.

```
...modyfikacja modułu changer.py bez zatrzymywania Pythona...
```

```
% notepad changer.py
```

Teraz zmień przypisanie zmiennej globalnej `message`, a także zmodyfikuj ciało funkcji `printer`.

```
message = "Po edycji"

def printer():
    print('przeładowany:', message)
```

Warto pamiętać: przeładowywane moduły

Poza możliwością przeładowywania (i tym samym wykonywania) modułów w sesji interaktywnej, funkcja `reload` przydaje się również w większych systemach, w szczególności kiedy koszt restartu całej aplikacji jest zbyt duży. Na przykład, dobrymi kandydatami do dynamicznego przeładowywania modułów są serwery gier i systemy, które muszą po uruchomieniu łączyć się z serwerami za pośrednictwem sieci.

Mogliwość taka przydaje się również w pracy z graficznym interfejsem użytkownika (wywołania zwrotne widgetu można zmienić w czasie, gdy GUI działa), a także kiedy Python wykorzystywany jest jako język osadzony w programie napisanym w C czy C++ (program go zawierający może zażądać przeładowania wykonywanego kodu w Pythonie bez konieczności zatrzymywania się). Więcej informacji na temat przeładowywania wywołań zwrotnych GUI oraz osadzonego kodu Pythona można znaleźć w książce *Programming Python*.

Przeładowywane pozwalają programom udostępniać bardzo dynamiczne interfejsy. Python jest na przykład często wykorzystywany jako język pozwalający na *dostosowanie większych systemów do własnych potrzeb*. Użytkownicy mogą dostosować produkty do swoich upodobań, pisząc fragmenty kodu w Pythonie na miejscu, bez konieczności ponownej komplikacji całego produktu (a nawet bez posiadania dostępu do jego kodu

źródłowego). W takim świecie już sam kod napisany w Pythonie dodaje programowi dynamiki.

Aby być jeszcze bardziej dynamicznymi, takie systemy mogą automatycznie i regularnie ładować napisany w Pythonie kod dostosowujący ich działanie. W ten sposób zmiany wprowadzane przez użytkownika są uwzględniane w czasie działania systemu — nie ma konieczności zatrzymywania go i ponownego uruchamiania za każdym razem, gdy napisany w Pythonie kod jest modyfikowany. Nie wszystkie systemy wymagają tak dynamicznego podejścia, jednak dla tych, które to robią, przeładowywanie modułów jest łatwym w użyciu narzędziem umożliwiającym dynamiczne dostosowanie aplikacji do własnych potrzeb.

Następnie powróć do okna Pythona i przeładuj moduł w celu pobrania nowego kodu. W tym przykładzie warto zwrócić uwagę na to, że ponowne zimportowanie modułu nie daje żadnego rezultatu — nadal otrzymujemy oryginalny komunikat, mimo że plik się zmienił. Aby aktywować aktualną wersję, musimy wywołać funkcję `reload`.

```
...powrót do interaktywnej sesji Pythona...
>>> import changer
>>> changer.printer()                                # Brak efektu – wykorzystuje
załadowany moduł
Pierwsza wersja
>>> from imp import reload
>>> reload(changer)                                 # Wymusza załadowanie i
wykonanie nowego kodu
<module 'changer' from 'changer.py'>
>>> changer.printer()                             # Teraz wykonuje nową wersję
przeładowany: Po edycji
```

Warto zauważyć, że funkcja `reload` tak naprawdę *zwraca* obiekt modułu — jej wynik jest zazwyczaj ignorowany, jednak ponieważ wynik wyrażenia jest wyświetlany w sesji interaktywnej, Python pokazuje domyślną reprezentację `<module 'nazwa'...>`.

Dwie ostatnie uwagi: po pierwsze, jeżeli użyjesz funkcji `reload`, prawdopodobnie będziesz chciał sparować ją z instrukcją `import`, a nie `from`, ponieważ nazwy importowane za pomocą instrukcji `from` nie są aktualizowane przez operacje przeładowania, co może pozostawić je w stanie, który jest wystarczająco dziwny, aby uzasadnić odłożenie dalszej dyskusji na ten temat do sekcji, w której omawiamy „pułapki” na programistów; znajdziesz ją na końcu rozdziału 25. Po drugie wywołanie funkcji `reload` aktualizuje tylko *jeden* moduł, ale utworzenie funkcji, która dokona przeładowania wszystkich niezbędnych modułów jest bardzo proste — rozbudowanie takiej funkcji pozostawimy do studium przypadku pod koniec rozdziału 25.

Podsumowanie rozdziału

Niniejszy rozdział skupił się na omówieniu podstawowych zagadnień związanych z tworzeniem kodu modułów — instrukcji `import` oraz `from` i wywołania funkcji `reload`. Nauczyliśmy się, że instrukcja `from` dokłada po prostu dodatkowy etap kopiujący zmienne z zainportowanego pliku, a także że funkcja `reload` wymusza ponowne zainportowanie pliku bez zatrzymywania i ponownego uruchamiania Pythona. Zapoznaliśmy się również z koncepcjami związanymi z przestrzeniami nazw i zobaczyliśmy, co dzieje się w przypadku zagnieżdżenia operacji

importowania. Wiemy także, w jaki sposób pliki stają się przestrzeniami nazw modułów, a także jakie są potencjalne pułapki związane z wykorzystywaniem instrukcji `from`.

Choć dowiedziałeś się już wystarczająco dużo, by korzystać z modułów we własnych programach, w kolejnym rozdziale rozszerzymy naszą wiedzę dotyczącą modelu importowania, prezentując *importowanie pakietów* — metodę pozwalającą na podanie w instrukcji `import` fragmentu ścieżki do katalogu zawierającego pożądany moduł. Jak zobaczymy, importowanie pakietów daje nam hierarchię przydatną w większych systemach i pozwala rozwiązać konflikty pomiędzy modułami o tych samych nazwach. Zanim jednak przejdziemy dalej, czas zająć się krótkim quizem podsumowującym omówione tutaj zagadnienia.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób tworzymy moduły?
2. W jaki sposób instrukcja `from` powiązana jest z instrukcją `import`?
3. W jaki sposób funkcja `reload` powiązana jest z operacją importowania?
4. Kiedy musimy użyć instrukcji `import` zamiast `from`?
5. Podaj trzy potencjalne pułapki związane z używaniem instrukcji `from`.
6. Jaka... jest prędkość lotu jaskółki bez obciążenia?

Sprawdź swoją wiedzę — odpowiedzi

1. Aby utworzyć moduł, powinieneś utworzyć plik tekstowy zawierający instrukcje języka Python. Każdy plik kodu źródłowego automatycznie staje się modułem i nie istnieje osobna składnia służąca do deklaracji modułu. Operacje importowania ładują pliki modułów do obiektów modułów znajdujących się w pamięci. Moduł można również utworzyć, pisząc kod w języku zewnętrznym, takim jak C lub Java, jednak tego typu moduły rozszerzeń pozostają poza zakresem niniejszej książki.
2. Instrukcja `from` importuje cały moduł, podobnie do instrukcji `import`, jednak dodatkowo kopiuje jeszcze jedną lub większą liczbę zmiennych z importowanego modułu do zasięgu, w którym się pojawia. Pozwala to na bezpośrednie używanie zimportowanych zmiennych (na przykład zmiennej *nazwa*) w miejsce konieczności uwzględniania w wyrażeniu nazwy modułu (w formie `moduł.nazwa`).
3. Domyslnie moduł jest importowany tylko raz na proces. Funkcja `reload` wymusza ponowne zimportowanie modułu. Jest to wykorzystywane przede wszystkim w celu uzyskania nowszej wersji kodu źródłowego modułu w czasie programowania, a także w sytuacjach wymagających dynamicznego dostosowywania aplikacji do potrzeb użytkownika.
4. Instrukcji `import` używa się zamiast `from` jedynie wtedy, gdy musimy uzyskać dostęp do zmiennych o tej samej nazwie występujących w dwóch różnych modułach. Ponieważ musimy w takiej sytuacji dodatkowo podać nazwę modułu zawierającego zmienną, nazwy te będą unikalne. Rozszerzenie `as` również może spowodować, że instrukcja `from` stanie się użyteczna w tym kontekście.

5. Instrukcja `from` może zaciemnić znaczenie zmiennej (to, w którym module została zdefiniowana), może powodować problemy w czasie wywoływania funkcji `reload` (zmienna może odnosić się do poprzedniej wersji obiektów), a także może uszkadzać przestrzeń nazw (po cichu nadpisując zmienne wykorzystywane w zakresie bieżącym). Forma `from *` jest pod wieloma względami gorsza — może poważnie zniszczyć przestrzeń nazw i zaciemnić znaczenie zmiennych. Najlepiej jest używać jej oszczędnie.

6. Jakiej jaskółki? Afrykańskiej czy europejskiej?

[1] Niektóre języki programowania działają inaczej i udostępniają *zasięgi dynamiczne*, gdzie zasięgi naprawdę mogą być uzależnione od wywołań w czasie wykonywania. Sprawia to jednak, że kod staje się bardziej zagmatwany, ponieważ znaczenie zmiennej może się z czasem zmienić. W Pythonie zasięgi odpowiadają praktycznie lokalizacji w kodzie programu.

Rozdział 24. Pakiety modułów

Dotychczas kiedy importowaliśmy moduły, ładowaliśmy pliki. To typowy model użycia modułów i technika, której w początkach naszej kariery programisty Pythona będziemy używać najczęściej. Importowanie modułów to jednak coś więcej, niż dotychczas sugerowałem.

Poza nazwą modułu w operacji importowania można również wymienić ścieżkę do katalogu. Katalog z kodem Pythona nazywa się *pakietem*, dlatego tego typu operacje importowania znane są jako *importowanie pakietów* (ang. *package import*). W rezultacie importowanie pakietów zamienia katalog z naszego komputera na kolejną przestrzeń nazw Pythona, z atrybutami odpowiadającymi podkatalogom oraz plikom modułów znajdujących się w tym katalogu.

Jest to opcja nieco bardziej zaawansowana, ale udostępniana przez nią hierarchia okazuje się przydatna do organizowania plików w większe systemy i zazwyczaj upraszcza ustawienia ścieżki wyszukiwania modułów. Jak zobaczymy, importowanie pakietów jest czasami wymagane do rozwiązania problemów z operacjami importowania powstających, kiedy na jednym komputerze zainstalowanych jest kilka plików programów o tej samej nazwie.

W niniejszym rozdziale omówimy również wprowadzony w najnowszych wersjach Pythona mechanizm i składnię *importów względnych*, który jest ściśle powiązany z pakietami i modułami. Jak się przekonasz, model ten modyfikuje ścieżki wyszukiwania w wersji 3.x i rozszerza instrukcję `from` w zakresie importowania nazw z pakietów zarówno w wersji 2.x, jak i 3.x. Taki model może sprawić, że importowanie wewnętrz pakietów będzie bardziej wyraźne i zwięzłe, ale związany jest z pewnymi kompromisami, które mogą wpływać na Twoje programy.

Dla użytkowników korzystających z Pythona 3.3 i nowszych wersji omówimy także nowy model *przestrzeni nazw pakietu*, który pozwala pakietom obejmować wiele katalogów i nie wymaga pliku inicjującego. Ten nowy model pakietów jest opcjonalny i może być używany w połączeniu z oryginalnym (lub jak kto woli „zwykłym”) modelem pakietu i rozszerza niektóre podstawowe koncepcje i reguły oryginalnego modelu. Z tego powodu najpierw przeanalizujemy tutaj zwykłe pakiety, a nowy model przestrzeni nazw pakietu przedstawimy jako temat opcjonalny.

Podstawy importowania pakietów

Na poziomie podstawowym importowanie pakietów jest całkiem proste — w miejscu, w którym w instrukcji `import` normalnie wstawiamy nazwę pliku, umieszczamy ścieżkę nazw rozdzielonych od siebie kropkami:

```
import dir1.dir2.mod
```

Tak samo wygląda to w przypadku instrukcji `from`.

```
from dir1.dir2.mod import x
```

Ścieżka z kropkami w tych instrukcjach ma odpowiadać ścieżce w systemie plików prowadzącej do pliku `mod.py` (lub podobnego — rozszerzenia mogą być różne). Powyższe instrukcje wskazują zatem, że na naszym komputerze istnieje katalog `dir1`, a w nim podkatalog `dir2` zawierający plik modułu `mod.py` (lub podobny).

Co więcej, takie operacje importowania sugerują, że katalog `dir1` znajduje się w katalogu nadzewnętrznym `dir0`, dostępnym dla ścieżki wyszukiwania modułów Pythona. Innymi słowy,

powyższe instrukcje sugerują, że w systemie plików istnieje struktura przypominająca poniższą (z separatorami w postaci lewych ukośników stosowanych w systemie Windows).

```
dir0\dir1\dir2\mod.py                                # Lub mod.pyc, mod.so i tak  
dalej
```

Katalog nadzędny *dir0* musi być dodany do ścieżki wyszukiwania modułów (o ile nie jest katalogiem głównym dla pliku najwyższego poziomu) — dokładnie tak samo, jakby *dir1* był plikiem modułu.

Mówiąc bardziej formalnie, pierwszy od lewej element ścieżki importu jest nazwą względną w ramach ścieżki wyszukiwania `sys.path`, którą mieliśmy okazję poznać w rozdziale 22. Od tego miejsca do końca ścieżki instrukcje `import` z naszego skryptu w jawnym sposobie określają ścieżki katalogów prowadzących do modułów.

Pakiety a ustawienia ścieżki wyszukiwania

Jeżeli korzystamy z tej opcji, należy pamiętać, że ścieżki katalogów w instrukcjach `import` mogą być tylko zmiennymi rozdzielonymi kropkami. Nie można w tych instrukcjach użyć żadnej składni ścieżek specyficznej dla określonej platformy, takiej jak `C:\dir1\Moje dokumenty\dir2` czy `../dir1` — takie ścieżki nie będą działały. Zamiast tego składni specyficznej dla platformy należy użyć w ustawieniach ścieżki wyszukiwania modułów i określić w ten sposób nazwę katalogu nadzędznego.

W poprzednim przykładzie katalog *dir0* — nazwa katalogu dodawana do ścieżki wyszukiwania modułów — może być dowolnie długą, specyficzną dla platformy ścieżką do katalogu, prowadzącą do katalogu *dir1*. Zamiast używać niepoprawnej instrukcji, takiej jak poniższa:

```
import C:\mycode\dir1\dir2\mod                      # Błąd – niepoprawna składnia
```

powinieneś dodać katalog `C:\mycode` do zmiennej środowiskowej `PYTHONPATH` lub pliku `.pth`, a następnie wpisać w kodzie programu następujące polecenie:

```
import dir1.dir2.mod
```

W rezultacie wpisy ze ścieżki wyszukiwania modułów będą zawierały *prefiksy* katalogów specyficzne dla platformy i prowadzące do nazw znajdujących się po lewej stronie instrukcji `import` lub `from`. Takie instrukcje importu same z siebie dostarczają pozostałą część ścieżki katalogu w sposób neutralny dla platformy^[1].

Jeżeli chodzi o proste importowanie plików, nie musisz dodawać katalogu nadzędznego *dir0* do ścieżki wyszukiwania modułów, jeżeli już tam jest — zgodnie z tym, co pokazywaliśmy w rozdziale 22., będzie to katalog główny pliku najwyższego poziomu, katalog, w którym pracujesz interaktywnie, standardowy katalog biblioteki lub katalog główny instalacji pakietów zewnętrznych. Tak czy inaczej, ścieżka wyszukiwania modułów musi zawierać wszystkie katalogi znajdujące się z lewej strony argumentu instrukcji importowania pakietu kodu.

Pliki pakietów `_init_.py`

Jeżeli zdecydujesz się na importowanie pakietów, musisz pamiętać o jeszcze jednym ograniczeniu, którego należy przestrzegać: przynajmniej do wersji 3.3 Pythona każdy katalog wymieniony w ścieżce instrukcji importowania pakietu musi zawierać plik o nazwie `_init_.py`, w przeciwnym razie operacja importowania zakończy się niepowodzeniem. Oznacza to, że w przykładzie wykorzystanym wyżej oba katalogi, *dir1* oraz *dir2*, muszą zawierać plik o nazwie `_init_.py`. Katalog nadzędny *dir0* nie musi zawierać tego pliku, ponieważ nie jest on wymieniony w samej instrukcji `import`.

Z formalnego punktu widzenia, jeżeli struktura katalogów wygląda następująco:

```
dir0\dir1\dir2\mod.py
```

a instrukcja `import` ma następującą postać:

```
import dir1.dir2.mod
```

to zastosowanie mają poniższe reguły:

- katalogi `dir1` oraz `dir2` muszą zawierać plik `__init__.py`,
- katalog nadzędny `dir0` nie musi zawierać pliku `__init__.py`; jeżeli plik ten będzie się w nim znajdował, zostanie zignorowany,
- katalog `dir0`, a nie `dir0\dir1`, musi być umieszczony w ścieżce wyszukiwania modułów `sys.path`.

Aby spełnić dwie pierwsze reguły, deweloperzy pakietów muszą utworzyć odpowiednie pliki, które omówimy już za chwilę. Aby spełnić ostatnią regułę, katalog `dir0` musi być składnikiem automatycznej ścieżki wyszukiwania (czyli musi znajdować się w katalogu domowym użytkownika, katalogu bibliotek lub katalogu `site-packages`) lub musi zostać umieszczony w zmiennej `PYTHONPATH`, pliku `.pth` albo ręcznie dodany do ścieżki `sys.path`.

W efekcie struktura katalogów tego przykładu powinna wyglądać następująco (wcięcia oznaczają zagnieżdżenia katalogów):

```
dir0\                                # Katalog w ścieżce wyszukiwania
modułów

    dir1\
        __init__.py
        dir2\
            __init__.py
            mod.py
```

Pliki `__init__.py` mogą zawierać kod Pythona, podobnie do normalnych plików modułów. Ich nazwy są specjalne, ponieważ zapisany w nich kod jest uruchamiany automatycznie przy pierwszym zimportowaniu katalogu przez program, a zatem służą przede wszystkim jako punkty zaczepienia do uruchomienia inicjalizacji wymaganych przez pakiet. Pliki te mogą być również zupełnie puste, a czasami pełnią także dodatkowe role, o czym opowiem w kolejnym podrozdziale.



Jak zobaczymy pod koniec tego rozdziału, od wersji 3.3 Pythona zniesiono wymóg posiadania przez paczki pliku o nazwie `__init__.py`. W tej wersji i późniejszych katalogi modułów bez takiego pliku można importować jako składające się z jednego katalogu *pakiety przestrzeni nazw*, które działają tak samo, ale nie uruchamiają kodu inicjalizującego. Przed wersją 3.3 i we wszystkich wersjach 2.x pakiety nadal wymagały plików `__init__.py`. Jak pokażemy za chwilę, w wersji 3.3 i późniejszych użycie takich plików wpływa również na zwiększenie wydajności.

Role pliku inicjalizacji pakietu

Pliki `__init__.py` służą jako punkty zaczepienia dla działań odbywających się w czasie inicjalizowania pakietów, deklarują katalogi jako pakiety Pythona, generując przestrzeń nazw dla katalogów i implementując zachowanie instrukcji `from *` (na przykład `from ... import *`), kiedy wykorzystuje się je w połączeniu z importowaniem pakietów.

Inicjalizacja pakietów

Za pierwszym razem, gdy Python importuje coś za pomocą katalogu, automatycznie wykonuje cały kod z pliku `__init__.py` tego katalogu. Z tego powodu pliki te są naturalnym miejscem do wstawienia kodu inicjalizującego stan wymagany przez pakiet, który może na przykład wykorzystać plik inicjalizujący do utworzenia wymaganych plików z danymi czy otwarcia połączenia z bazą danych. Zazwyczaj pliki `__init__.py` nie są przydatne, jeżeli zostaną wykonane bezpośrednio; są one uruchamiane automatycznie przy pierwszym dostępie do pakietu.

Deklaracje użyteczności modułu

Jednym z zadań plików `__init__.py` jest również zadeklarowanie, że dany katalog jest pakietem Pythona. Obecność tych plików zapobiega niezamierzonemu ukrywaniu prawdziwych modułów przez podobne nazwy katalogów, które pojawiają się w ścieżce wyszukiwania modułów. Bez tego zabezpieczenia Python mógłby wybrać katalog, który nie ma nic wspólnego z Twoim kodem, tylko dlatego, że pojawia się wcześniej w ścieżce wyszukiwania. Jak zobaczymy później, pakiety przestrzeni nazw Pythona 3.3 w znacznym stopniu redukują tę rolę, ale uzyskują podobny efekt w sposób algorytmiczny, skanując ścieżkę w poszukiwaniu kolejnych plików.

Inicjalizacja przestrzeni nazw modułu

W modelu importowania pakietów ścieżki katalogów ze skryptu stają się po zimportowaniu prawdziwymi ścieżkami zagnieżdżonych obiektów. Jak widać w poprzednim przykładzie, po zimportowaniu wyrażenie `dir1.dir2.mod` działa i zwraca obiekt modułu, którego przestrzeń nazw zawiera wszystkie zmienne przypisane przez plik `__init__.py` katalogu `dir2`. Takie pliki udostępniają przestrzeń nazw dla obiektów modułów tworzonych dla katalogów, które w przeciwnym razie nie miałyby żadnego skojarzonego pliku modułu.

*Zachowanie instrukcji `from *`*

Jako zaawansowaną opcję możemy wykorzystać listy `__all__` z plików `__init__.py` do zdefiniowania, co jest eksportowane, kiedy katalog importowany jest za pomocą instrukcji `from *`. W pliku `__init__.py` lista `__all__` ma być listą nazw podmodułów, które powinny zostać zimportowane, kiedy instrukcji `from *` użyjemy na nazwie pakietu (katalogu). Jeżeli lista `__all__` nie zostanie zdefiniowana, instrukcja `from *` nie załaduje automatycznie podmodułów zagnieżdżonych w katalogu. Zamiast tego załaduje tylko zmienne zdefiniowane przez przypisania w pliku `__init__.py` katalogu, w tym wszystkie podmoduły w jawnym sposobie zimportowane przez kod tego pliku. Na przykład użycie instrukcji `from submodule import X` w pliku `__init__.py` katalogu sprawia, że zmienna `X` z podmodułu `submodule` będzie dostępna w przestrzeni nazw tego katalogu (dodatkowe zastosowania listy `__all__` omówimy w rozdziale 25.; służy ona również do deklarowania eksportów `from *` z prostych plików).

Pliki `__init__.py` możemy również po prostu pozostawić puste, jeżeli ich rola wykracza poza nasze potrzeby (i szczerze mówiąc, w praktyce te pliki są najczęściej puste). Muszą jednak istnieć, aby importowanie katalogów w ogóle działało.

	Nie powinieneś mylić plików <code>__init__.py</code> w pakietach z metodami konstruktora klasy <code>__init__</code> , które poznamy w następnej części książki. Pierwsze z nich są modułami ładowanymi przez interpreter w ramach importu pakietu, drugie są metodami wywoływanymi w celu utworzenia instancji klasy. Oba komponenty mają role inicjujące, ale znacznie różnią się od siebie.
---	--

Przykład importowania pakietu

Utwórzmy teraz prawdziwy kod przykładu, o jakim mówiliśmy, w celu pokazania, w jaki sposób działa inicjalizacja plików oraz ścieżek. Poniższe trzy pliki zapisane są w katalogu *dir1* oraz jego podkatalogu *dir2* — w komentarzach zamieszczamy pełne ścieżki do tych plików:

```
# Plik dir1\__init__.py
print ('dir1 init')
x = 1

# Plik dir1\dir2\__init__.py
print ('dir2 init')
y = 2

# Plik dir1\dir2\mod.py
print ('w pliku mod.py')
z = 3
```

Katalog *dir1* będzie tutaj albo bezpośrednim podkatalogiem naszego katalogu roboczego (np. katalogu domowego), albo bezpośrednim podkatalogiem katalogu wymienionego w ścieżce wyszukiwania modułów (czyli z technicznego punktu widzenia w *sys.path*). W obu sytuacjach katalog nadzędny *dir1* nie musi zawierać pliku *__init__.py*.

Instrukcje *import* wykonują pliki inicjalizacyjne każdego katalogu przy pierwszym przejściu tego katalogu, w miarę jak Python schodzi w dół ścieżki; instrukcje *print* zostały dodane w celu ułatwienia śledzenia sposobu działania plików:

```
C:\code> python                                #Uruchamiamy w katalogu
nadzędnym dir1

>>> import dir1.dir2.mod                      # Pierwszy import wykonuje pliki
inicjalizacyjne

dir1 init

dir2 init

w mod.py

>>>

>>> import dir1.dir2.mod                      # Kolejne importy tego nie robią
```

Tak jak w przypadku plików modułów, zaimportowany już katalog może zostać przekazany do funkcji *reload* w celu wymuszenia ponownego wykonania tego elementu. Jak widać poniżej, funkcja *reload* akceptuje ścieżki z kropkami, by móc przeładować zagnieżdżone katalogi oraz pliki.

```
>>> from imp import reload                    # instrukcja from jest niezbędna
tylko w wersji 3.x

>>> reload(dir1)

dir1 init

<module 'dir1' from '.\\dir1\\__init__.py'>

>>>

>>> reload(dir1.dir2)

dir2 init
```

```
<module 'dir1.dir2' from '.\\dir1\\dir2\\__init__.py'>
```

Po zimportowaniu ścieżka z instrukcji `import` staje się ścieżką zagnieżdzonego obiektu w skrypcie. W kodzie przedstawionym poniżej, `mod` jest obiektem zagnieżdżonym w obiekcie `dir2`, który jest z kolei zagnieżdżony w obiekcie `dir1`.

```
>>> dir1
<module 'dir1' from '.\\dir1\\__init__.py'>
>>> dir1.dir2
<module 'dir1.dir2' from '.\\dir1\\dir2\\__init__.py'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from '.\\dir1\\dir2\\mod.py'>
```

Tak naprawdę każda nazwa katalogu ze ścieżki staje się zmienną przypisaną do obiektu modułu, którego przestrzeń nazw inicjalizowana jest przez wszystkie przypisania z pliku `__init__.py` tego katalogu. Zmienna `dir1.x` odnosi się do zmiennej `x` przypisanej w pliku `dir1__init__.py`, tak samo jak zmienna `mod.z` odnosi się do zmiennej `z` przypisanej w pliku `mod.py`.

```
>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

Instrukcja `from` a instrukcja `import` w importowaniu pakietów

Instrukcje `import` mogą być nieco niewygodne w połączeniu z pakietami, ponieważ często musimy w programie wpisywać te same ścieżki. W przykładzie wyżej za każdym razem, gdy chcemy dotrzeć do zmiennej `z`, musimy ponownie wpisać i wykonać pełną ścieżkę od katalogu `dir1`. Jeżeli próbujemy uzyskać bezpośredni dostęp do katalogu `dir2`, otrzymamy błąd.

```
>>> dir2.mod
NameError: name 'dir2' is not defined
>>> mod.z
NameError: name 'mod' is not defined
```

Często w przypadku pakietów wygodniejsze jest zatem skorzystanie z instrukcji `from`, co pozwala uniknąć ponownego wpisywania całych ścieżek przy każdym dostępie do obiektów. Co jednak ważniejsze, jeżeli kiedykolwiek zmienimy strukturę drzewa katalogów, instrukcja `from` wymaga uaktualnienia tylko jednej ścieżki w kodzie, podczas gdy `import` może wiązać się w większą liczbą zmian. Omówione w kolejnym rozdziale rozszerzenie `import as` może również być pomocne, gdyż podaje krótszy synonim pełnej ścieżki i zapewnia wygodny sposób zmiany nazwy, kiedy taka sama nazwa pojawia się w wielu modułach.

```
C:\\code> python
```

```
>>> from dir1.dir2 import mod          # Ścieżka podana jedynie tutaj
dir1 init
dir2 init
w pliku mod.py

>>> mod.z                            # Nie powtarzamy ścieżki
3

>>> from dir1.dir2.mod import z
>>> z
3

>>> import dir1.dir2.mod as mod      # Użycie krótszej nazwy (zobacz
rozdział 25.)

>>> mod.z
3

>>> from dir1.dir2.mod import z as modz    #To samo, gdy nazwy ze sobą
kolidują (zobacz rozdział 25.)
>>> modz
3
```

Do czego służy importowanie pakietów

Osoby zaczynające swoją przygodę z Pythonem powinny przed przejściem do pakietów opanować proste moduły, ponieważ pakiety są już mechanizmem nieco bardziej zaawansowanym. Pełnią jednak użyteczne role, w szczególności w większych programach — sprawiają, że operacje importowania są bardziej informatywne, służą jako narzędzia organizacyjne, upraszczają ścieżkę wyszukiwania modułów i mogą rozwiązać różne niejasności.

Przede wszystkim jednak, ponieważ importowanie pakietów udostępnia pewne informacje o katalogach w plikach programów, ułatwia lokalizację plików i służy także jako narzędzie organizacyjne. Bez ścieżek pakietów często musielibyśmy się odwoływać do ścieżki wyszukiwania pakietów, aby odnaleźć określone pliki. Co więcej, jeżeli zorganizujemy swoje pliki w podkatalogi zgodne z pewnymi obszarami funkcjonalnymi, importowanie pakietów sprawia, że bardziej oczywiste staje się, jaką rolę pełni moduł, dzięki czemu kod jest bardziej czytelny. Na przykład normalne zaimportowanie pliku z katalogu znajdującego się w ścieżce wyszukiwania modułów, takie jak poniższe:

```
import utilities
```

oferuje nam o wiele mniej informacji niż operacja importowania uwzględniająca ścieżkę:

```
import database.client.utilities
```

Importowanie pakietów może również znacznie uprościć nasze ustawienia zmiennej `PYTHONPATH` oraz plików `.pth`. W rzeczywistości, jeżeli korzystasz z jawnego importowania pakietów i dokonujesz importu pakietów względem wspólnego katalogu głównego, w którym przechowywany jest cały kod Twojego programu, tak naprawdę potrzebujesz tylko jednego wpisu na ścieżce wyszukiwania: wspólnego katalogu głównego. Wreszcie importowanie

pakietów służy rozwiązywaniu niejednoznaczności importów poprzez wyraźne określenie, które pliki chcesz zimportować, i rozwiązuje konflikty, gdy ta sama nazwa modułu pojawia się w więcej niż jednym miejscu. W kolejnym podrozdziale zajmiemy się bardziej szczegółowo tą właśnie rolą.

Historia trzech systemów

Jedyna sytuacja, w której importowanie pakietów jest *wymagane* do rozwiązania niejasności, pojawia się, kiedy na jednym komputerze zainstalowana jest większa liczba programów z plikami noszącymi te same nazwy. Jest to problem instalacyjny, jednak w praktyce może stać się dotkliwy — szczególnie biorąc pod uwagę tendencję programistów do używania prostych i podobnych nazw plików modułów. Aby go zilustrować, zajmiemy się pewnym hipotetycznym scenariuszem wydarzeń.

Załóżmy, że programista tworzy w Pythonie program zawierający plik o nazwie *utilities.py* (ze wspólnym kodem narzędziowym), a także plik najwyższego poziomu *main.py* wykorzystywany przez użytkowników do uruchomienia programu. W całym programie pliki wykorzystują instrukcję `import utilities` do załadowania wspólnego kodu narzędzi. Kiedy program jest dostarczany klientom, stanowi jedno archiwum *.tar* czy *.zip* zawierające wszystkie pliki programu, a po instalacji rozpakowuje wszystkie pliki do jednego katalogu na komputerze docelowym o nazwie *system1*.

```
system1\  
    utilities.py          # Wspólne funkcje i klasy  
    narzędzi  
  
    main.py              # Uruchamia program  
  
    other.py             # Importuje utilities w celu  
    załadowania narzędzi
```

Załóżmy teraz, że drugi programista tworzy inny program z plikami o nazwach *utilities.py* oraz *main.py* i ponownie wykorzystuje instrukcję `import utilities` w całym programie w celu załadowania pliku ze wspólnym kodem. Kiedy drugi system zostanie pobrany i zainstalowany na tym samym komputerze co pierwszy, jego pliki zostaną rozpakowane do nowego katalogu o nazwie *system2* na komputerze klienta, tak by nie nadpisały plików o tych samych nazwach z pierwszego systemu.

```
system2\  
    utilities.py          # Wspólne narzędzia  
    main.py              # Uruchamia program  
  
    other.py             # Importuje narzędzia
```

Jak na razie nie ma żadnych problemów — oba systemy mogą współistnieć i mogą być wykonywane na tym samym komputerze. Tak naprawdę nie musimy nawet konfigurować ścieżki wyszukiwania modułów, by skorzystać z obu programów — ponieważ Python zawsze najpierw przeszukuje katalog główny (czyli katalog zawierający plik najwyższego poziomu), operacje importowania w plikach obu systemów automatycznie zobaczą wszystkie pliki w katalogu danego systemu. Jeżeli na przykład klikniemy plik *system1\main.py*, wszystkie operacje importowania najpierw będą przeszukiwały katalog *system1*. W podobny sposób po uruchomieniu pliku *system2\main.py* jako pierwszy przeszukany zostanie katalog *system2*. Należy pamiętać, że ustawienia ścieżki wyszukiwania modułów są potrzebne tylko wtedy, gdy importujemy pliki pomiędzy katalogami.

Załóżmy jednak, że po zainstalowaniu tych dwóch programów na komputerze decydujemy się użyć jakiejś części kodu z każdego z plików *utilities.py* we własnym programie. W końcu jest to

wspólny kod narzędzi, a kod napisany w Pythonie z natury służy do ponownego użycia. W takim przypadku możesz użyć poniższych instrukcji w kodzie pliku utworzonego w trzecim katalogu, tak by załadować jeden z dwóch plików z narzędziami.

```
import utilities  
utilities.func('mielonka')
```

I teraz zaczynamy dostrzegać problem. Aby taki kod działał, musimy ustawić ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona katalogi zawierające pliki *utilities.py*. Który katalog należy jednak umieścić jako pierwszy — *system1* czy *system2*?

Problemem jest *liniowa* natura ścieżki wyszukiwania, która zawsze jest przeglądana od lewej do prawej strony, więc bez względu na to, jak długo byśmy się nad tym zastanawiali, zawsze otrzymamy plik *utilities.py* z katalogu wymienionego jako pierwszy (bardziej na lewo) w ścieżce wyszukiwania. W takiej postaci nigdy nie będziemy w stanie zainportować tego pliku z innego katalogu.

Możemy spróbować zmodyfikować *sys.path* w skrypcie przed każdą operacją importowania, ale to dodatkowa operacja, na dodatek bardzo podatna na błędy, a zmiana ustawień zmiennej *PYTHONPATH* przed każdym uruchomieniem programu w Pythonie jest zbyt nużąca i nie pozwala na używanie obu wersji w jednym pliku. Domyślne rozwiązanie sprawia, że jesteśmy w kropce.

Problem ten może rozwiązać właśnie importowanie pakietów. Zamiast instalować programy w niezależnych katalogach wymienionych na ścieżce wyszukiwania modułów osobno, możesz spakować je i zainstalować w *podkatalogach* we wspólnym katalogu głównym. Możemy na przykład zorganizować cały kod tego przykładu w postaci hierarchii katalogów, wyglądającej następująco:

```
root\  
    system1\  
        __init__.py  
        utilities.py  
        main.py  
        other.py  
  
    system2\  
        __init__.py  
        utilities.py  
        main.py  
        other.py  
  
    system3          # Tutaj lub w dowolnym innym miejscu  
        __init__.py      #Plik __init__.py jest tutaj potrzebny, gdy  
        # pakiet jest importowany w innym miejscu  
        myfile.py       # Tutaj nasz nowy kod
```

W takiej sytuacji do ścieżki wyszukiwania dodajemy tylko wspólny katalog główny. Jeżeli wszystkie operacje importowania w naszym kodzie są wykonywane względem wspólnego katalogu głównego, możemy zainportować plik narzędzi z *dowolnego* programu za pomocą importowania pakietu — nazwa katalogu zawierającego plik sprawia, że ścieżka (i tym samym referencja do modułu) staje się unikalna. Tak naprawdę możemy nawet zainportować *oba* pliki

narzędzi w tym samym module, dopóki wykorzystujemy instrukcję `import` i powtarzamy pełną ścieżkę za każdym razem, gdy odwołujemy się do modułów narzędzi.

```
import system1.utilities  
import system2.utilities  
system1.utilities.function('mielonka')  
system2.utilities.function('jajka')
```

Nazwa modułu zawierającego plik sprawia, że referencja do modułu staje się unikalna.

Warto zauważyć, że w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from` tylko wtedy, gdy musimy uzyskać dostęp do tego samego atrybutu z dwóch lub większej liczby ścieżek. Gdyby nazwa wywoływanej funkcji była w każdej ścieżce inna, można by było użyć instrukcji `from`, co pozwalałoby uniknąć powtarzania pełnej ścieżki za każdym wywołaniem którejś z funkcji, tak jak opisano to wcześniej; do utworzenia unikalnych synonimów nazw możemy również użyć rozszerzenia `as`.

Zwrócić również uwagę, że w pokazanej wcześniej hierarchii zainstalowanych plików pliki `_init_.py` zostały dodane do katalogów `system1` oraz `system2`, tak by importowanie pakietów działało, jednak nie znalazły się w *katalogu głównym*. Obecność tych plików jest wymagana jedynie w katalogach wymienionych w instrukcjach `import`. Jak pamiętamy, pliki `_init_.py` są wykonywane automatycznie przez Pythona przy pierwszym importowaniu pakietów.

Z technicznego punktu widzenia w tym przypadku podkatalog `system3` nie musi się znajdować w *katalogu głównym* — dotyczy to jedynie pakietów kodu, z których będziemy importować. Ponieważ jednak nigdy nie wiemy, kiedy nasze własne moduły będą mogły się przydać innym programom, możemy również dobrze od razu umieścić je we wspólnym katalogu głównym w celu uniknięcia problemów z konfliktami między takimi samymi nazwami w przyszłości.

Wreszcie warto zauważyć, że instrukcje importowania z obu oryginalnych systemów działają bez zmian. Ponieważ najpierw przeszukiwane są ich *katalogi domowe*, dodanie wspólnego katalogu do ścieżki wyszukiwania nie ma znaczenia dla kodu z katalogów `system1` oraz `system2`. Nadal można w nich zastosować polecenie `import utilities` i oczekwać, że odnajdą w ten sposób własne pliki — choć nie można tego zrobić, jeżeli zostaną użyte jako pakiety w wersji 3.x, co wyjaśnimy w kolejnej sekcji. Co więcej, jeżeli będziemy konsekwentnie rozpakowywać wszystkie programy napisane w Pythonie w jednym katalogu głównym, tak jak zaprezentowano to powyżej, konfiguracja ścieżki staje się banalnie prosta — wystarczy do niej tylko raz dodać wspólny katalog główny.

Warto pamiętać: pakiety modułów

Ponieważ pakiety są standardową częścią Pythona, powszechnie stosowaną praktyką jest to, że większe rozszerzenia firm trzecich są dostarczane jako zestawy katalogów z pakietami, a nie płaskie listy modułów. Na przykład pakiet rozszerzeń `win32all` dla Pythona był jednym z pierwszych, które były udostępniane w postaci pakietów. Wiele jego modułów narzędziowych znajduje się w pakietach importowanych ze ścieżkami. Na przykład, aby załadować narzędzia COM po stronie klienta, powinieneś użyć następującej instrukcji:

```
from win32com.client import constants, Dispatch
```

Przedstawiony wiersz kodu pobiera nazwy z modułu `client` pakietu `win32com` — z podkatalogu instalacyjnego.

Import pakietów jest również wszechobecny w kodzie uruchamianym w implementacji Jython, opartej na języku Java, ponieważ biblioteki Java są również zorganizowane w hierarchie. W najnowszych wydaniach Pythona narzędzia poczty elektronicznej oraz XML są podobnie zorganizowane w podkatalogach pakietów w standardowej bibliotece, a w Pythonie 3.x jeszcze więcej modułów zostało powiązanych w pakiety, w tym narzędzia

takie jak graficzny interfejs użytkownika `tkinter`, narzędzia sieciowe `HTTP` i inne. Przykładowe polecenia pokazane poniżej importują kilka różnych narzędzi bibliotecznych (dla wersji 3.x; sposób użycia w wersji 2.x może się różnić):

```
from email.message import Message  
from tkinter.filedialog import askopenfilename  
from http.server import CGIHTTPRequestHandler
```

Niezależnie od tego, czy samodzielnie będziesz tworzył katalogi pakietów, czy nie, to prawdopodobnie wcześniej czy później i tak będziesz z nich importował potrzebne Ci narzędzia.

Względne importowanie pakietów

Dotychczas podczas omawiania zagadnień związanych z importowaniem pakietów koncentrowaliśmy się głównie na importowaniu plików *spoza* pakietu. Wewnątrz pakietu importowanie plików tego samego pakietu może korzystać z tej samej składni z pełnymi ścieżkami, co importowanie plików spoza pakietu — i jak zobaczymy, czasami nawet powinno. Pliki pakietów mogą jednak również korzystać ze specjalnych, uproszczonych reguł importowania *wewnątrz pakietu*, gdzie zamiast określania pełnej ścieżki do modułu w pakiecie, można zastosować ścieżkę *względną* wewnątrz pakietu.

Sposób działania tego mechanizmu jest zależny od wersji: Python 2.x podczas importowania domyślnie przeszukuje katalogi pakietów, podczas gdy wersja 3.x do importowania z katalogu pakietu wymaga jawnego podania względnej ścieżki pliku w pakiecie. Ta zmiana w wersji 3.x może poprawić czytelność kodu, czyniąc importowanie plików z tego samego pakietu bardziej oczywistym, ale jest jednocześnie niezgodna z wersją 2.x i może spowodować niepoprawne działanie niektórych programów.

Jeżeli rozpoczynasz swoją przygodę z Pythonem od wersji 3.x, powinieneś skupić się na nowej składni i modelu importowania. Jeżeli jednak używałeś już wcześniej starszych wersji Pythona, z pewnością będziesz zainteresowany tym, co zmieniło się w modelu importowania zaimplementowanym w wersji 3.x. Zacznijmy zatem nasze rozważania od tego drugiego tematu.



Jak się przekonasz w tej sekcji, użycie importowania względnego w pakietach może faktycznie ograniczyć role plików. Krótko mówiąc, nie będzie można ich już używać jako plików programów wykonywalnych w wersjach 2.x i 3.x. Z tego powodu w wielu przypadkach lepszym rozwiązaniem mogą być normalne, pełne ścieżki importowania pakietów. Niemniej ten nowy mechanizm znalazzł zastosowanie w wielu programach Pythona i z pewnością zasługuje na to, aby programiści Pythona się z nim zapoznali i lepiej zrozumieli zarówno jego zalety, jak i wady.

Zmiany w Pythonie 3.0

Sposób działania mechanizmu importowania wewnętrz pakietów uległ drobnym zmianom w Pythonie 3.x. Ta zmiana dotyczy wyłącznie importowania plików, które są częścią tego samego pakietu i znajdują się w jego katalogu. Importowanie z innych plików działa tak samo jak dotychczas. W przypadku importowania wewnętrz pakietów w Pythonie 3.x wprowadzono następujące zmiany:

- Zmodyfikowano mechanizm importowania w taki sposób, aby własny katalog pakietu był pomijany. Importowanie sprawdza jedynie ścieżki znajdujące się w ścieżce wyszukiwania `sys.path`. Tego typu import nazywamy *bezwzględnym* (ang. *absolute*).
- Rozszerzono składnię instrukcji `from`, pozwalając na jawne żądanie przeszukiwania ścieżki tylko wewnątrz bieżącego pakietu; możemy to zrobić za pomocą wiodącej kropki w ścieżce pakietu. Tego typu import nazywamy *względnym* (ang. *relative*).

Opisane dwie zmiany obowiązują od wersji 3.x Pythona. Nowa składnia importowania względnego za pomocą `from` jest również dostępna w Pythonie 2.x, ale zmiana mechanizmu domyślnej ścieżki wyszukiwania musi być włączona jako opcja. Włączenie tej opcji może jednak spowodować niepoprawne działanie programów dla wersji 2.x, ale jest dostępne w celu zapewnienia kompatybilności w przód z wersją 3.x.

Skutkiem tej zmiany jest konieczność użycia w Pythonie 3.x (i opcjonalnie w 2.x) specjalnej składni polecenia `from` z kropką wiodącą w ścieżce, pozwalającej na zimportowanie modułów znajdujących się w tym samym pakiecie, chyba że polecenie importu zawiera pełną ścieżkę względem katalogu głównego pakietów w `sys.path` lub polecenia importu są podawane względem zawsze przeszukiwanego katalogu domowego programu (który jest zwykle również bieżącym katalogiem roboczym).

Domyślnie jednak katalog pakietu nie jest automatycznie przeszukiwany, a importowanie wewnątrz pakietu plików znajdujących się bezpośrednio w katalogu pakietu nie powiedzie się bez użycia specjalnej składni polecenia `from`. Jak się niebabem przekonasz, w wersji 3.x może to wpływać na sposób strukturyzacji importów lub katalogów modułów przeznaczonych do użycia zarówno w programach najwyższego poziomu, jak i pakietach do importowania. Najpierw jednak przyjrzyjmy się dokładniej, jak to wszystko działa.

Podstawy importowania względnego

Zarówno w Pythonie 3.x, jak i 2.x instrukcje `from` mogą wykorzystywać wiodące kropki `(.)`, które sygnalizują, że wymagane moduły są zlokalizowane w tym samym pakiecie (nazywamy to *importem względnym*), a nie w dowolnym miejscu ścieżki wyszukiwania (co nazywamy *importem bezwzględnym*). A dokładniej:

- *Importowanie z użyciem kropek wiodących*: zarówno w Pythonie 3.x, jak i 2.x można użyć wiodącej kropki w instrukcjach `from`, aby wymusić import *względny* w ramach pakietu: tego typu importy wyszukują moduły tylko w katalogu bieżącego pakietu i nie znajdą modułów o tych samych nazwach zapisanych w innych miejscach ścieżki wyszukiwania (`sys.path`). W efekcie moduły danego pakietu mogą przeciązać moduły zewnętrzne.
- *Importowanie bez użycia kropek wiodących*: w Pythonie 2.x importowanie bez wiodącej kropki powoduje wyszukiwanie w *trybie względnym*, a następnie *bezwzględnym*, z tym, że wyszukiwanie odbywa się w pierwszej kolejności w bieżącym pakiecie. W Pythonie 3.x natomiast importy w ramach pakietu są domyślnie *bezwzględne* — jeżeli nie zastosowano wiodącej kropki, import pomija moduły pakietu i wyszukuje wyłącznie w ścieżce `sys.path`.

Na przykład zarówno w Pythonie 3.x, jak i 2.x można zastosować następującą instrukcję:

```
from . import spam # Import względny w stosunku do pakietu
```

Jej wykonanie spowoduje zimportowanie modułu o nazwie `spam`, położonego w tym samym katalogu pakietu, co moduł, w którym występuje ta instrukcja. Podobnie instrukcja:

```
from spam import name
```

oznacza „z modułu `spam` znajdującego się w tym samym katalogu zainportuj zmienną o nazwie `name`”.

Zachowanie instrukcji importu *bez wiodących kropek* jest różne w różnych wersjach Pythona. W 2.x import tego typu domyślnie powoduje wyszukiwanie *względne* (czyli w katalogu bieżącego pakietu), a następnie, jeżeli się ono nie powiedzie, *bezwzględne* w ramach ścieżki wyszukiwania, chyba że na początku pliku importującego jako pierwszą instrukcję wykonywalną umieścimy następujące polecenie:

```
from __future__ import absolute_import      # Użyj w wersji 2.x względnego
modelu importowania z wersji 3.x
```

Jeżeli takie polecenie występuje, włącza mechanizm wyszukiwania bezwzględnego znanego z wersji 3.x. Po włączeniu tej opcji w wersjach 3.x i 2.x import bez wiodącej kropki w nazwie modułu zawsze powoduje, że Python pomija względne komponenty ścieżki wyszukiwania importu modułu i zamiast tego szuka wyłącznie w katalogach zawartych w `sys.path`. Na przykład poniższa instrukcja w wersji 3.x spowoduje zimportowanie modułu `string` ze ścieżki `sys.path`, a nie załadowanie modułu `string` zdefiniowanego w bieżącym pakiecie:

```
import string                                # Pominięcie wersji zdefiniowanej
w bieżącym pakiecie
```

Dla kontrastu: bez użycia instrukcji `from __future__` w wersji 2.x Python w pierwszej kolejności podejmie próbę zimportowania modułu z bieżącego pakietu, jeżeli w pakiecie znajduje się lokalny moduł `string`. Aby uzyskać ten efekt w wersji 3.x oraz w 2.x przy włączonej opcji importu bezwzględnego, należy jawnie zastosować składnię importu względnego:

```
from . import string                          # Wyszukuje w bieżącym pakiecie
```

Ta instrukcja działa obecnie zarówno w Pythonie 2.x, jak i 3.x. Różnica między wersjami Pythona polega jedynie na tym, że w 3.x zastosowanie tej składni jest *konieczne* do załadowania modułu z bieżącego katalogu, gdy dany plik jest częścią pakietu (chyba że podane zostaną pełne ścieżki pakietów).

Zwrót uwagę, że wiodące kropki mogą być użyte do wymuszenia względnego importowania jedynie w instrukcji `from`, nie w instrukcji `import`. W Pythonie 3.x instrukcja `import nazwa_modułu` zawsze działa w trybie bezwzględnym, czyli pomija bieżący katalog pakietu. W wersji 2.x taka instrukcja nadal wykonuje import względny, najpierw przeszukując katalog pakietu. Instrukcje `from` bez kropek wiodących zachowują się tak samo jak instrukcje `import` — bezwzględnie tylko w 3.x (pomijając katalog pakietu) i względnie, a następnie bezwzględnie w wersji 2.x (najpierw przeszukując katalog pakietu).

Możliwe są również inne względne wzorce odwołań oparte na kropkach. W pliku modułu znajdująącym się w katalogu pakietu o nazwie `mypkg` następujące alternatywne formy importu działają tak, jak to opisano w komentarzach:

```
from .string import name1, name2  # Importuje nazwy z mypkg.string
from . import string                # Importuje mypkg.string
from .. import string               # Importuje string z tego samego poziomu,
na którym znajduje się mypkg
```

Aby lepiej wyjaśnić działanie tych przykładów i ułatwić zrozumienie przyczyny wdrażania tej dodatkowej złożoności w mechanizmie importowania, musimy dokonać krótkiej dygresji.

Do czego służą importy względne

Oprócz tego, że importowanie wewnętrz tego samego pakietu stało się bardziej przejrzyste, mechanizm importowania względnego został zaprojektowany między innymi po to, aby umożliwić skryptom rozwiązywanie dwuznaczności, które mogą powstawać, gdy plik o tej samej

nazwie pojawia się w wielu miejscach ścieżki wyszukiwania modułów. Rozważmy następującą strukturę pakietu:

```
mypkg\  
  __init__.py  
  main.py  
  string.py
```

W ten sposób zdefiniowany jest pakiet `mypkg` zawierający moduły `mypkg.main` i `mypkg.string`. Założymy teraz, że moduł główny próbuje zaimportować moduł o nazwie `string`. W wersji 2.x i nowszych Python zacznie wyszukiwanie w katalogu `mypkg`, realizując *import względny*. Znajdzie zapisany w nim plik o nazwie `print.py` i zaimportuje go, przypisując nazwę `string` w przestrzeni nazw `mypkg.main` pakietu `mypkg`.

Może się jednak okazać, że intencją programisty było zaimportowanie modułu `print` standardowej biblioteki Pythona. Niestety, w starszych wersjach Pythona nie ma prostego sposobu na zignorowanie modułu `mypkg.string` i wymuszenie importu z biblioteki standardowej lub innego modułu znajdującego się w ścieżce wyszukiwania. Co więcej, tego problemu nie możemy również rozwiązać, wykorzystując ścieżki importowania w pakiecie, ponieważ nie możemy zakładać, że układ ścieżek biblioteki standardowej będzie taki sam na każdej maszynie.

Innymi słowy, proste importy w pakietach mogą wprowadzać niejednoznaczności i zwiększać podatność na błędy: w ramach pakietu nie wiadomo, czy instrukcja `import spam` odnosi się do modułu w pakiecie, czy poza nim. W efekcie lokalny moduł lub pakiet może przesłonić (celowo lub przypadkowo) inny, znajdujący się w ścieżce wyszukiwania `sys.path`.

W praktyce użytkownicy Pythona unikają stosowania dla własnych modułów nazw zdefiniowanych w standardowej bibliotece (jeżeli potrzebujesz standardowego modułu `string`, nie powinieneś tworzyć własnego modułu o tej samej nazwie!). Jednak to nie wystarczy, jeżeli w pakiecie zostanie przypadkowo przesłonięty moduł standardowej biblioteki; co więcej, w nowej wersji Pythona mogą pojawić się nowe moduły o nazwach użytych przez nas w programie. Kod wykorzystujący importowanie względne jest też trudniejszy do zrozumienia, ponieważ użytkownik analizujący kod programu może nie mieć pewności co do tego, który pakiet miał być zaimportowany. W kodzie zawsze lepiej jest jednoznacznie dać do zrozumienia, jakie były intencje programisty.

Importowanie względne w wersji 3.x

Aby rozwiązać ten dylemat, mechanizm importowania wewnętrz pakietów zmieniono w Pythonie 3.x i ma teraz charakter wyłącznie bezwzględny (można to również włączyć jako opcję w wersji 2.x). W tym modelu instrukcja `import` w naszym przykładowym pliku `mypkg/main.py` zawsze znajdzie moduł `string` poza pakietem poprzez importowanie bezwzględne ze ścieżki wyszukiwania `sys.path`:

```
import string          # Importuje moduł string spoza pakietu  
(import bezwzględny)
```

Importy z użyciem instrukcji `from` ze ścieżkami bez wiodących kropek również traktowane są jako bezwzględne.

```
from string import name      # Importuje name z modułu string poza  
                             pakietem
```

Jeżeli chcesz zaimportować moduł zdefiniowany w pakiecie bez podawania jego pełnej ścieżki od katalogu głównego, możesz skorzystać z importowania względnego, umieszczając kropkę w instrukcji `from`:

```
from . import string          # Importuje mypkg.string (import względny)
```

Taka forma powoduje zimportowanie modułu `string` zdefiniowanego w bieżącym pakiecie i jest względnym odpowiednikiem bezwzględnej postaci importu z poprzedniego przykładu (oba polecenia ładują moduł jako całość). W przypadku zastosowania tej składni importu wyszukiwanie jest wykonywane wyłącznie w katalogu pakietu.

Składni importu względnego można użyć również do zainportowania nazw z modułu:

```
from .string import name1, name2      # Importuje nazwy z mypkg.string
```

Ta instrukcja odwołuje się do modułu `string` zdefiniowanego w bieżącym pakiecie. Jeżeli ten kod wystąpi w module `mypkg.main`, nastąpi import nazw `name1` i `name2` z `mypkg.string`.

Pojedyncza kropka w imporcie względnym efektywnie oznacza *bieżący* pakiet, czyli katalog, w którym zapisany jest plik, w którym zadeklarowano `import`. Dodatkowa kropka spowoduje względny import, rozpoczynając od katalogu *nadrzędnego*, na przykład:

```
from .. import spam           # Importuje spam sąsiadujący z mypkg
```

Powyższy kod spowoduje zainportowanie pakietu `spam` zdefiniowanego na tym samym poziomie, co `mypkg`. Mówiąc bardziej ogólnie, kod znajdujący się w module `A.B.C` może używać dowolnej z następujących form:

```
from . import D               # Importuje A.B.D      (. oznacza A.B)
from .. import E              # Importuje A.E       (... oznacza A)
from .D import X              # Importuje A.B.D.x   (. oznacza A.B)
from ..E import X              # Importuje A.E.x     (... oznacza A)
```

Względne importy a bezwzględne ścieżki pakietów

W module można też wskazać pełną ścieżkę do pakietu, wykorzystując składnię importów bezwzględnych. W poniższym przykładzie pakiet `mypkg` zostanie wczytany ze ścieżki wyszukiwania `sys.path`:

```
from mypkg import string        # Importuje mypkg.string (import bezwzględny)
```

Mechanizm ten opiera swoje działanie na konfiguracji i zdefiniowanej kolejności ścieżek, natomiast w przypadku importów względnych nie ma tego typu niejednoznaczności. Co więcej, taki import jak przedstawiony w przykładzie wymaga, aby katalog, w którym zdefiniowany jest pakiet `mypkg`, był zadeklarowany w ścieżce wyszukiwania. Prawdopodobnie dzieje się tak, jeżeli `mypkg` jest katalogiem głównym pakietu (w przeciwnym razie pakiet nie mógłby być użyty z zewnątrz!), ale ten katalog może być zagnieźdzony w znacznie większym drzewie pakietów. Jeżeli `mypkg` nie jest katalogiem głównym pakietu, instrukcje bezwzględnego importu muszą zawierać pełną ścieżkę do pakietu od głównego katalogu w ścieżce wyszukiwania `sys.path`.

```
from system.section.mypkg import string    # system znajduje się w katalogu zdefiniowanym w sys.path
```

W przypadku złożonych lub głęboko zagnieźdzonych pakietów pełna ścieżka będzie dłuża, co może wymagać wpisywania znacznie większej ilości kodu niż w przypadku zastosowania importowania względnego z kropką:

```
from . import string            # Składnia importów względnych
```

W tym ostatnim przykładzie pakiet bieżący jest przeszukiwany automatycznie, niezależnie od ustawień ścieżki wyszukiwania, kolejności ścieżki wyszukiwania i zagnieźdzania katalogu. Z drugiej strony bezwzględna forma pełnej ścieżki będzie działać bez względu na to, w jaki

sposób plik jest używany — jako część programu czy pakietu — o czym opowiemy już w kolejnym podrozdziale.

Zasięg importów względnych

Importy względne mogą przy pierwszym kontakcie wydać się dość skomplikowanym zagadnieniem, ale wszystko znacznie się upraszcza po poznaniu pewnych podstawowych zasad:

- **Importy względne są stosowane wyłącznie wewnętrz pakietów.** Pamiętaj, że zmiana ścieżki wyszukiwania modułu dotyczy tylko instrukcji importowania w plikach modułu używanych jako część pakietu — czyli inaczej mówiąc, dotyczy importowania *wewnętrz pakietu*. Normalne importy w plikach, które nie są częścią pakietu, nadal działają dokładnie tak, jak opisano wcześniej, automatycznie przeszukując najpierw katalog zawierający skrypt najwyższego poziomu.
- **W względne importy stosuje się wyłącznie w instrukcji from.** Pamiętaj też, że nowa składnia importowania dotyczy tylko instrukcji `from` i nie ma zastosowania do instrukcji `import`. Importy względne rozpoznawane są po tym, że po słowie kluczowym `from` następuje jedna lub więcej kropek. Nazwy modułów zawierające kropki, ale bez kropki wiodącej są traktowane jako zwykły import pakietów, a nie import względny.

Innymi słowy: względne importy z pakietów w wersji 3.x to w zasadzie rezygnacja ze stosowanej w Pythonie 2.x reguły wyszukiwania modułów w pakietach oraz dodanie składni wymuszającej wyszukiwanie względne w pakięcie. Jeżelipisałeś kod dla Pythona tak, aby nie korzystać z wyszukiwania względnego, charakterystycznego dla wersji 2.x (na przykład zawsze podawałeś pełną ścieżkę od katalogu głównego do modułu), to zmiany wprowadzone w Pythonie 3.x raczej nie będą stanowiły zagrożenia z punktu widzenia kompatybilności. Jeżeli tego nie zrobiłeś, musisz zaktualizować pliki pakietu, aby korzystać z nowej składni polecenia `from` dla lokalnych plików pakietów lub używać pełnych ścieżek bezwzględnych.

Podsumowanie reguł wyszukiwania modułów

W przypadku pakietów i importowania względnego reguły wyszukiwania modułów w Pythonie 3.x, które omawialiśmy do tej pory, można streszczyć w następujący sposób:

- Podstawowe moduły z prostymi nazwami (np. A) są wyszukiwane w każdym katalogu z listy `sys.path`, od lewej do prawej. Ta lista jest budowana z domyślnych ustawień systemowych i uzupełniana o ustawienia konfigurowane przez użytkownika.
- Pakiety są po prostu katalogami zawierającymi moduły Pythona oraz specjalny plik `__init__.py`, który umożliwia stosowanie w importach ścieżki typu `A.B.C`. W takim importie katalog A powinien znajdować się w ścieżce wyszukiwania `sys.path`, B to podkatalog katalogu A, natomiast C jest modułem lub inną nazwą importowaną z B.
- W plikach pakietów zwykłe instrukcje `import` oraz `from` stosują tę samą regułę z użyciem ścieżki wyszukiwania `sys.path`. Importy w pakietach wykorzystujące instrukcję `from` oraz ścieżkę modułu rozpoczętą się od kropki stosują specjalną regułę importów względnych, to znaczy nazwa jest wyszukiwana wyłącznie w odniesieniu do pakietu, a wyszukiwanie w ścieżce `sys.path` nie jest wykonywane. Na przykład w instrukcji `from . import A` wyszukiwanie modułu A będzie realizowane wyłącznie w katalogu zawierającym plik, w którym zdefiniowano tę instrukcję.

Python 2.x działa tak samo, z tym wyjątkiem, że normalne importowanie ze ścieżką bez kropek wiodących również automatycznie najpierw przeszukuje *katalog pakietów*, zanim przejdzie do `sys.path`.

Podsumowując, importowanie w Pythonie wybiera między *względnymi* (w zawierającym katalogu) i *bezwzględnymi* (w katalogu ze ścieżki wyszukiwania `sys.path`) operacjami w

następujący sposób:

Importowanie z kropką: from . import m, from .m import x

Jest względne zarówno w wersji 2.x, jak i 3.x.

Importowanie bez kropki: import m, from m import x

Jest najpierw względne, a następnie bezwzględne w wersji 2.x, a bezwzględne tylko w wersji 3.x.

Jak zobaczymy później, Python 3.3 dodaje kolejny element do modułów — *pakiety przestrzeni nazw* (ang. *namespace packages*) — który jest w dużej mierze odmienny od historii importowania względnego, tutaj omawianej. Ten nowszy model również obsługuje importowanie względne wewnątrz pakietów i jest po prostu innym sposobem na zbudowanie pakietu. Pakiety przestrzeni nazw usprawniają procedurę wyszukiwania importu, umożliwiając rozłożenie zawartości pakietu na wiele prostych katalogów, ale później pakiet jako całość zachowuje się tak samo w kwestii względnych reguł importu.

Importy względne w działaniu

Wystarczy tej teorii, pokażemy zatem kilka prostych przykładów, aby zademonstrować w praktyce koncepcję importów względnych.

Importowanie spoza pakietów

Przede wszystkim, jak już wspominaliśmy wcześniej, mechanizm importów względnych nie ingeruje w możliwość importowania spoza pakietów. Dzięki temu poniższy kod importujący moduł `string` standardowej biblioteki Pythona zadziała zgodnie z oczekiwaniami:

```
C:\code> c:\Python33\python
>>> import string
>>> string
<module 'string' from 'C:\\\\Python33\\\\lib\\\\string.py'>
```

Jeżeli jednak do katalogu, w którym pracujemy, dodamy moduł o nazwie `string`, to załadowany zostanie ten moduł, ponieważ na pierwszym miejscu w ścieżce wyszukiwania znajduje się bieżący katalog roboczy (ang. *CWD — current working directory*).

```
# code\string.py
print('string' * 8)
C:\code> c:\Python33\python
>>> import string
stringstringstringstringstringstringstring
>>> string
<module 'string' from '.\\\\string.py'>
```

Innymi słowy, zwykłe importy nadal są względne w stosunku do katalogu „domowego” (czyli katalogu, w którym zapisany jest skrypt, lub katalogu, z którego został uruchomiony). W rzeczywistości składnia importów względnych nie jest dozwolona w kodzie znajdującym się w pliku niebędącym częścią pakietu.

```
>>> from . import string
```

```
SystemError: Parent module '' not loaded, cannot perform relative import
```

W tej sekcji kod wprowadzany w sesji interaktywnej zachowuje się tak samo, jakby był uruchamiany w skrypcie najwyższego poziomu, ponieważ pierwszym wpisem w ścieżce `sys.path` jest albo interaktywny katalog roboczy, albo katalog zawierający plik najwyższego poziomu. Jedyna różnica polega na tym, że pierwszy element ścieżki `sys.path` jest katalogiem bezwzględnym, a nie pustym ciągiem znaków:

```
# code\main.py

import string
pliku # Ten sam kod, ale w

print(string)

C:\code> C:\python33\python main.py # Wyniki w wersji 2.x
będą takie same

stringstringstringstringstringstringstringstring

<module 'string' from 'C:\\code\\string.py'>
```

Jak widać, próba wykonania polecenia `from . import string` w pliku niebędącym częścią pakietu kończy się niepowodzeniem, tak samo jak w przypadku sesji interaktywnej.

Importy wewnętrz pakietów

Usuń teraz lokalny moduł `string`, który zapisaliśmy w bieżącym katalogu roboczym, i zbuduj tam katalog pakietu zawierający dwa moduły, w tym wymagany, ale pusty plik `code\pkg__init__.py`. Główne katalogi pakietów omawianych w tej sekcji znajdują się w bieżącym katalogu roboczym, dodawanym automatycznie do ścieżki `sys.path`, więc nie musimy ustawiać zmiennej `PYTHONPATH`. Ze względu na oszczędność miejsca w dużej mierze pominimy też puste pliki `__init__.py` i większość komunikatów o błędach (a użytkownicy systemów innych niż Windows będą musieli samodzielnie przetłumaczyć pokazane niżej polecenia powłoki na swoją platformę):

```
C:\code> del string* # w wersjach 3.2+ usuwamy pliki kodu bajtowego
__pycache__\string*

C:\code> mkdir pkg

C:\code> notepad pkg\__init__.py

# code\pkg\spam.py

import eggs # <== Działa w wersji 2.x, ale nie w 3.x!
print(eggs.x)

# code\pkg\eggs.py

X = 99999

import string

print(string)
```

W pierwszym module tego pakietu próbujemy zaimportować drugi za pomocą zwykłej instrukcji `import`. W wersji 2.x Python potraktuje takie polecenie jako import względny, ale w wersji 3.x jako bezwzględny, stąd w tym drugim przypadku próba wykonania tego polecenia zakończy się niepowodzeniem. Innymi słowy, w wersji 2.x najpierw przeszukiwany jest pakiet modułu, w którym wykonywany jest import, ale w wersji 3.x tak się nie dzieje. Jest to jeden ze szczególnów implementacji Pythona 3.x, który *nie jest kompatybilny* z poprzednimi wersjami, i należy mieć ten fakt na uwadze.

```
C:\code> c:\Python27\python
>>> import pkg.spam
<module 'string' from 'c:\Python27\lib\string.pyc'>
99999
C:\code> c:\Python33\python
>>> import pkg.spam
ImportError: No module named 'eggs'
```

Aby nasze moduły działały prawidłowo zarówno w wersji 2.x, jak i 3.x, powinieneś zmodyfikować instrukcję importu w pierwszym z plików w taki sposób, by wykorzystać składnię importów względnych i wskazać Pythonowi, aby modułu `eggs` szukał również w katalogu pakietu (dla wersji 3.x):

```
# code\pkg\spam.py
from . import eggs           # <== użycie względnych importów w 2.x i 3.x
print(eggs.x)

# code\pkg\eggs.py
X = 99999
import string
print(string)

C:\code> c:\Python27\python
>>> import pkg.spam
<module 'string' from 'c:\Python27\lib\string.pyc'>
99999
C:\code> c:\Python33\python
>>> import pkg.spam
<module 'string' from 'c:\Python33\lib\string.py'>
99999
```

Importy są nadal względne w stosunku do bieżącego katalogu roboczego

Zwróć uwagę na to, że moduły w pakietach ciągle mają dostęp do modułów biblioteki standardowej, takich jak `string`, jednak ich normalne importy są nadal względne w stosunku do ścieżki wyszukiwania. W rzeczywistości, jeżeli do katalogu roboczego dodamy moduł `string`, to przy próbie importu zostanie załadowany właśnie on, nie moduł biblioteki standardowej. Choć w wersji 3.x możesz pominąć katalog pakietu za pomocą importu bezwzględnego, nie ma możliwości pominięcia katalogu głównego programu, który importuje taki pakiet:

```
# code\string.py
print('string' * 8)
# code\pkg\spam.py
from . import eggs
```

```

print(eggs.x)

# code\pkg\eggs.py

X = 99999

import string          # <== Znajduje moduł string w katalogu roboczym,
nie w bibliotece standardowej!

print(string)

C:\code> c:\Python33\python    # W wersji 2.x wynik będzie taki sam

>>> import pkg.spam

stringstringstringstringstringstringstringstring

<module 'string' from '.\\string.py'>

99999

```

Użycie importów względnych i bezwzględnych

Aby pokazać, w jaki sposób zasady wyszukiwania modułów wpływają na użycie modułów biblioteki standardowej, ponownie zresetujemy nasz pakiet. Usuń lokalny moduł `string` i utwórz nowy, ale wewnątrz pakietu.

```

C:\code> del string*          # w wersjach 3.2+ usuwamy pliki kodu bajtowego
__pycache__\string*

# code\pkg\spam.py

import string          # <== Względny w wersji 2.x, bezwzględny w
wersji 3.x

print(string)

# code\pkg\string.py

print('Ni' * 8)

```

To, która wersja modułu `string` zostanie zaimportowana, zależy od użytej wersji Pythona. Jak już pokazywaliśmy, wersja 3.x import w pierwszym pliku traktuje jako bezwzględny, pomijając moduły pakietu, ale w 2.x tak się nie dzieje — to kolejny przykład braku *kompatybilności wstępnej* wersji 3.x.

```

C:\code> c:\Python33\python

>>> import pkg.spam

<module 'string' from 'C:\\Python33\\lib\\string.py'>

C:\code> c:\Python27\python

>>> import pkg.spam

NiNiNiNiNiNiNiNi

<module 'pkg.string' from 'pkg\string.py'>

```

Użycie składni importów względnych w wersji 3.x wymusza przeszukiwanie pakietu podobnie jak w wersji 2.x, ale dzięki możliwości użycia w wersji 3.x importów bezwzględnych lub względnych mamy pełną kontrolę nad tym, który pakiet zostanie zaimportowany. W rzeczywistości *to właśnie jest przyczyną wprowadzenia takiej zmiany w Pythonie 3.x*.

```
# code\pkg\spam.py
```

```
from . import string          # <== Import względny zarówno w wersji 2.x,  
jak i 3.x  
  
print(string)  
  
# code\pkg\string.py  
print('Ni' * 8)  
  
C:\code> c:\Python33\python  
->>> import pkg.spam  
  
NiNiNiNiNiNiNiNi  
  
<module 'pkg.string' from '.\\pkg\\string.py'>  
  
C:\code> c:\Python27\python  
->>> import pkg.spam  
  
NiNiNiNiNiNiNiNi  
  
<module 'pkg.string' from 'pkg\string.py'>
```

Importy względne przeszukują tylko pakiety

Zwróć uwagę na to, że składnia importów względnych jest w rzeczywistości *deklaracją wiązania*, nie jedynie właściwością. Jeżeli usuniemy plik *string.py* i cały powiązany z nim kod bajtowy z naszego przykładu, import względny zadeklarowany w pliku *spam.py* nie uda się ani w wersji 3.x, ani w 2.x — Python nie podejmie próby zaimportowania modułu *string* ze standardowej ścieżki wyszukiwania (czyli z biblioteki standardowej ani z jakiejkolwiek innej zdefiniowanej w ścieżce).

```
# code\pkg\spam.py  
  
from . import string          # <== Nie działa ani w wersji 2.x, ani w 3.x;  
w pakiecie nie ma pliku string.py!  
  
C:\code> del pkg\string*  
  
C:\code> C:\Python33\python  
->>> import pkg.spam  
  
ImportError: cannot import name string  
  
C:\code> C:\Python27\python  
->>> import pkg.spam  
  
ImportError: cannot import name string
```

Moduły wskazywane przez importy względne muszą istnieć w katalogu pakietu.

Importy są nadal względne w stosunku do katalogu roboczego (cd.)

Importy bezwzględne pozwalają pominąć moduły pakietów, ale nadal są zależne od elementów ścieżki *sys.path*. W ostatnim teście zdefiniujemy dwa własne moduły *string*. Zapiszemy je w ten sposób, aby jeden z modułów był w pakiecie, drugi w bieżącym katalogu roboczym; trzeci moduł znajduje się w bibliotece standardowej Pythona.

```
# code\string.py
```

```

print('string' * 8)

# code\pkg\spam.py

from . import string           # <== Względny zarówno w wersji 2.x, jak i 3.x

print(string)

# code\pkg\string.py

print('Ni' * 8)

```

Gdy zaimportujemy moduł `string` z użyciem składni importów względnych, w obu wersjach Pythona otrzymamy moduł zapisany w pakiecie, dokładnie tak, jak się tego spodziewamy:

```

C:\code> c:\Python33\python      # Taki sam rezultat w 2.x

>>> import pkg.spam

NiNiNiNiNiNiNi

<module 'pkg.string' from '.\\pkg\\string.py'>

```

Gdy użyjemy składni importów bezwzględnych, wynik importu będzie ponownie zależny od wersji Pythona: w wersji 2.x import ten będzie względny, a w wersji 3.x „bezwzględny”, co w tym przypadku oznacza pominięcie wersji zdefiniowanej w pakiecie i użycie pakietu zdefiniowanego w ścieżce roboczej (wersja z biblioteki standardowej *nie będzie* użyta).

```

# code\string.py

print('string' * 8)

# code\pkg\spam.py

import string                  # <== Import względny w 2.x, bezwzględny w
3.x: katalog roboczy!

print(string)

# code\pkg\string.py

print('Ni' * 8)

C:\code> c:\Python33\python

>>> import pkg.spam

stringstringstringstringstringstringstring

<module 'string' from '.\\string.py'>

C:\code> c:\Python27\python

>>> import pkg.spam

NiNiNiNiNiNiNi

<module 'pkg.string' from 'pkg\\string.pyc'>

```

Jak widzimy, pakiety mogą deklarować import modułów z lokalnych pakietów, ale importy pozostają względne w stosunku do ścieżki wyszukiwania. W tym przypadku plik zdefiniowany w katalogu roboczym programu przesłania moduł biblioteki standardowej o tej samej nazwie. Zmiany mechanizmu wyszukiwania modułów wprowadzone w wersji 3.x realizują jedynie możliwość wyboru modułu z bieżącego pakietu lub spoza niego (w ramach importów względnych lub bezwzględnych). Jednak mechanizm importu jest zależny od środowiska, w

którym jest wykonywany, bezwzględne importy w wersji 3.x nadal nie są zabezpieczeniem przed efektami ubocznymi przesłonięcia nazw modułów przez inne zapisane w ścieżce wyszukiwania.

Warto przeprowadzić kilka eksperymentów z tymi przykładami w celu lepszego zrozumienia zasad importów. W rzeczywistości problemy z importami nie są tak powszechnne, jak można by się obawiać, wnioskując z przykładów: importy można ustrukturalizować, ścieżkę wyszukiwania można modyfikować do własnych potrzeb, aby importy działały dokładnie tak, jak tego oczekujemy. Należy jednak mieć na uwadze, że importy w bardziej skomplikowanych konfiguracjach systemowych mogą być zależne od kontekstu, a na protokół importu modułów ma wpływ jakość projektu biblioteki aplikacji.

Pułapki związane z importem względnym w pakietach: zastosowania mieszane

Teraz gdy znasz już sposoby działania importów względnych w pakietach, powinieneś również zapamiętać, że nie zawsze są one najlepszym wyborem. Bezwzględny import pakietów, z pełną ścieżką do katalogu w katalogu `sys.path`, nadal jest najczęściej preferowanym rozwiązańiem, zarówno w przypadku niejawnego importu względnego w Pythonie 2.x, jak i jawnego składni kropkowej importu względnego w Pythonie 2.x i 3.x. Początkowo może się to wydawać mało istotne, ale najprawdopodobniej szybko docenisz ważność tego zagadnienia po rozpoczęciu samodzielnego kodowania własnych pakietów.

Jak widzieliśmy, składnia importów względnych w Pythonie 3.x i domyślna reguła wyszukiwania bezwzględnego wymuszają jawnie importowanie wewnątrz pakietów, dzięki czemu kod staje się bardziej czytelny i łatwiejszy do zrozumienia, a co więcej, umożliwia także dokonywanie wyraźnego wyboru w niektórych scenariuszach konfliktu nazw. Są jednak również dwa główne następstwa tego modelu, o których należy pamiętać:

- Zarówno w Pythonie 3.x, jak i 2.x użycie instrukcji importu względnego dla pakietu niejawnie wiąże plik z katalogiem pakietu i rolą, wykluczając tym samym jego użycie w inny sposób.
- W Pythonie 3.x nowa zmiana reguły wyszukiwania względnego oznacza, że plik nie może już służyć jednocześnie jako moduł skryptu i moduł pakietu tak łatwo, jak to jest możliwe w wersji 2.x.

Przyczyny tych ograniczeń są nieco subtelne, ale dość oczywiste, biorąc pod uwagę następujące fakty:

- Ani wersja 3.x, ani 2.x Pythona nie pozwalają na używanie względnej instrukcji `from ..`, chyba że plik importujący jest częścią pakietu (inaczej mówiąc, moduł będący częścią pakietu jest importowany z innego miejsca pakietu).
- Podczas importowania Python 3.x nie przeszukuje własnego katalogu modułów pakietu, chyba że użyjemy względnej instrukcji `from .` (lub moduł znajduje się w bieżącym katalogu roboczym albo katalogu domowym głównego skryptu).

Korzystanie z importu względnego uniemożliwia tworzenie pakietów, które służą zarówno jako programy wykonywalne, jak i pakiety do importu z zewnętrz. Co więcej, niektóre pliki również nie mogą już służyć jednocześnie jako moduły skryptów i moduły pakietów. Jeżeli chodzi o instrukcje importu, reguły wyglądają następująco — pierwszy przykład dotyczy trybu *pakiet* w obu Pythonach, a drugi dotyczy trybu *program* tylko w wersji 3.x:

```
from . import mod          # Niedozwolone w trybie bez pakietu zarówno w  
wersji 2.x, jak i 3.x  
  
import mod                # W trybie pakietu nie przeszukuje własnego  
katalogu pliku (w wersji 3.x)
```

W efekcie w przypadku plików, które mają być używane w wersji 2.x lub 3.x, może być konieczne wybranie jednego trybu użycia — *pakietu* (z importem względnym) lub *programu* (z prostym importami) i wyodrębnienie prawdziwych plików modułów w osobnym podkatalogu, oddzielonym od plików skryptów najwyższego poziomu.

Alternatywnie możesz spróbować ręcznie wprowadzać zmiany w ścieżce `sys.path` (zadanie niewidoczne i podatne na błędy) lub zawsze używać pełnych ścieżek pakietów w importach bezwzględnych zamiast składni względnej lub prostych importów i zakładać, że katalog główny pakietu znajduje się w ścieżce wyszukiwania modułów:

```
from system.section.mypkg import mod          # Działa zarówno w trybie
program, jak i pakiet
```

Ze wszystkich opisanych schematów ten ostatni — import z pełnymi ścieżkami pakietu — może być najbardziej przenośny i funkcjonalny, ale musimy przejść do bardziej konkretnego kodu, aby przekonać się dlaczego.

Problem

Na przykład w Pythonie 2.x często stosuje się ten sam *pojedynczy katalog* jako program i pakiet, używając normalnego importu bez kropki. W trybie programu importy opierają się na katalogu domowym programu (skryptu), a w trybie pakietu (importy wewnętrz pakietu) najpierw przeszukiwany jest wzajemny katalog pakietu, a dopiero potem ścieżka bezwzględna. Nie działa to jednak w wersji 3.x — w trybie pakietu zwykłe importowanie nie ładuje już modułów znajdujących się w tym samym katalogu, chyba że katalog ten jest głównym katalogiem pakietu lub bieżącym katalogiem roboczym (i dlatego znajduje się na ścieżce `sys.path`).

Oto jak to wygląda w działaniu, z minimalną ilością kodu (dla zwięzości w tej sekcji ponownie pomijam pliki `__init__.py` wymagane dla pakietów przed wersją 3.3 Pythona, a dla wprowadzenia małego urozmaicenia używam programu uruchamiającego z systemu Windows, opisanego w dodatku B):

```
# code\pkg\main.py

import spam

# code\pkg\spam.py

import eggs          # <== Działa, jeżeli jest w katalogu "."
=, czyli katalogu domowym głównego skryptu

# code\pkg\eggs.py

print('Eggs' * 4)      # Ale plik nie zostanie załadowany,
jeżeli jest używany jako pakiet w 3.x!

c:\code> python pkg\main.py    # OK jako program, zarówno w 2.x, jak i w
3.x

EggsEggsEggsEggs

c:\code> python pkg\spam.py

EggsEggsEggsEggs

c:\code> py -2            # OK jako pakiet w 2.x; wyszukiwanie
wzajemne, a później bezwzględne

>>> import pkg.spam       # 2.x: proste importy przeszukują
najpierw katalog pakietów

EggsEggsEggsEggs
```

```
C:\code> py -3 # Ale wersja 3.x nie potrafi odnaleźć
tutaj pliku; pozostaje tylko import bezwzględny

>>> import pkg.spam # 3.x: proste importy przeszukują tylko
bieżący katalog roboczy plus ścieżkę sys.path

ImportError: No module named 'eggs'
```

Następnym krokiem mogłoby być dodanie *importu względnego* do użytku w wersji 3.x, ale tutaj to nie pomoże. W poniższym przykładzie zachowujemy pojedynczy katalog zarówno dla głównego skryptu, jak i modułów pakietu oraz dodajemy wymagane kropki — takie rozwiązanie działa zarówno w wersji 2.x, jak i 3.x, gdy katalog jest importowany jako pakiet, ale kończy się niepowodzeniem, gdy jest używany jako katalog programu (w tym podczas próby bezpośredniego uruchomienia modułu jako skryptu):

```
# code\pkg\main.py
import spam

# code\pkg\spam.py
from . import eggs # <== Nie jest pakietem, jeżeli znajduje się
tutaj skrypt główny

# code\pkg\eggs.py
print('Eggs' * 4)

c:\code> python # OK jako pakiet, ale nie jako program (w
wersjach 3.x i 2.x)

>>> import pkg.spam
EggsEggsEggsEggs

c:\code> python pkg\main.py
SystemError: ... cannot perform relative import

c:\code> python pkg\spam.py
SystemError: ... cannot perform relative import
```

Rozwiązanie nr 1: podkatalogi pakietów

W przypadku takiego mieszanego zastosowania jednym z rozwiązań jest izolacja wszystkich plików oprócz głównego skryptu w osobnym *podkatalogu* — w ten sposób importowanie modułów wewnętrz pakietu nadal działa we wszystkich Pythonach, możesz używać katalogu nadrzędnego jako samodzielnego programu, a katalog zagnieżdżony nadal może służyć jako pakiet do użytku dla innych programów:

```
# code\pkg\main.py
import sub.spam # <== Działa, jeżeli przeniesiemy moduły do
podkatalogu „poniżej” pliku głównego

# code\pkg\sub\spam.py
from . import eggs # Importy względne wewnętrz pakietu teraz
działają poprawnie (moduły w podkatalogach)

# code\pkg\sub\eggs.py
print('Eggs' * 4)
```

```
c:\code> python pkg\main.py      # Z głównego skryptu – ten sam rezultat
zarówno w wersji 2.x, jak i 3.x

EggsEggsEggsEggs

c:\code> python                  # Z innych programów – ten sam rezultat
zarówno w wersji 2.x, jak i 3.x

>>> import pkg.sub.spam

EggsEggsEggsEggs
```

Potencjalnym minusem takiego schematu jest to, że nie będzie można uruchamiać modułów pakietów bezpośrednio w celu sprawdzenia ich za pomocą odpowiedniego kodu testującego, choć zamiast tego wszelkie testy można osobno umieścić w katalogu nadzędnym:

```
c:\code> py -3 pkg\sub\spam.py  # Poszczególne moduły nie mogą być
uruchamiane indywidualnie, np. do testów

SystemError: ... cannot perform relative import
```

Rozwiążanie 2: import bezwzględny z użyciem pełnej ścieżki

Alternatywnie użycie *importu z podaniem pełnej ścieżki* również rozwiązałoby nasz problem — taki scenariusz wymaga co prawda, aby katalog nadzędny dla głównego katalogu pakietu był w ścieżce wyszukiwania, ale w praktyce nie jest to nic nadzwyczajnego dla rzeczywistych pakietów oprogramowania. Większość pakietów Pythona albo wymaga takiego ustawienia, albo zarządza jego automatyczną obsługą za pomocą narzędzi instalacyjnych (takich jak *distutils*, które pozwalają na przechowywanie pakietu w katalogu znajdującym się w domyślnej ścieżce wyszukiwania modułów, takim jak katalog główny pakietów witryny; więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 22.):

```
# code\pkg\main.py

import spam

# code\pkg\spam.py

import pkg.eggs                      # <== Pełne ścieżki działają we wszystkich
przypadkach; 2.x+3.x

# code\pkg\eggs.py

print('Eggs' * 4)

c:\code> set PYTHONPATH=C:\code

c:\code> python pkg\main.py      # Z głównego skryptu – ten sam rezultat
zarówno w wersji 2.x, jak i 3.x

EggsEggsEggsEggs

c:\code> python                  # Z innych programów – ten sam rezultat
zarówno w wersji 2.x, jak i 3.x

>>> import pkg.spam

EggsEggsEggsEggs
```

W przeciwieństwie do rozwiązań z podkatalogiem importy bezwzględne pełnej ścieżki, takie jak te pokazane powyżej, pozwalają również na samodzielne uruchamianie modułów w celu przetestowania:

```
c:\code> python pkg\spam.py      # Poszczególne moduły mogą być uruchamiane
indywidualnie w 2.x i 3.x
```

EggsEggsEggsEggs

Przykład: aplikacja z kodem autotestu modułu (wprowadzenie)

Na koniec przedstawimy jeszcze jeden przykład typowego problemu i jego pełną ścieżkę rozwiązania. Wykorzystuje ona powszechną technikę, którą omówimy w następnym rozdziale, ale sama koncepcja jest wystarczająco prosta, aby pokazać ją już tutaj (choćż możesz równie dobrze wrócić do tego przykładu nieco później).

Rozważ następujące dwa moduły w katalogu pakietu, z których drugi zawiera kod *autotestu*. Upraszczając, wartością atrybutu `__name__` tego modułu jest ciąg `__main__`, gdy jest uruchamiany jako skrypt najwyższego poziomu, ale nie podczas importowania, co pozwala na używanie tego modułu zarówno jako modułu, jak i skryptu:

```
# code\dualpkg\m1.py
def somefunc():
    print('m1.somefunc')

# code\dualpkg\m2.py
...tutaj import m1...           # Zamień ten wiersz na odpowiednie polecenie
import

def somefunc():
    m1.somefunc()
    print('m2.somefunc')

if __name__ == '__main__':
    somefunc()                  # Autotest lub kod skryptu najwyższego poziomu
```

Drugi moduł musi zaimportować ten pierwszy, w którym pojawia się wiersz zastępczy „*... tutaj import m1...*”. Zastąpienie tego wiersza względową instrukcją importu działa, gdy plik jest używany jako pakiet, ale nie jest dozwolone w trybie skryptu ani w wersji 2.x, ani w 3.x (wyniki i komunikaty o błędach zostały tutaj pominięte ze względu na oszczędność miejsca; zobacz plik `dualpkg\results.txt` w przykładach książki dla pełnego zestawu komunikatów):

```
# code\dualpkg\m2.py
from . import m1
c:\code> py -3
>>> import dualpkg.m2                # OK
C:\code> py -2
>>> import dualpkg.m2                # OK
c:\code> py -3 dualpkg\m2.py          # Nie działa!
c:\code> py -2 dualpkg\m2.py          # Nie działa!
```

I odwrotnie, prosta instrukcja importu działa w trybie skryptu zarówno w wersji 2.x, jak i 3.x, a nie działa w trybie pakietu tylko w wersji 3.x, ponieważ w tej wersji takie instrukcje nie przeszukują katalogu pakietów:

```
# code\dualpkg\m2.py
import m1
c:\code> py -3
```

```
>>> import dualpkg.m2 # Nie działa!  
c:\code> py -2  
>>> import dualpkg.m2 # OK  
c:\code> py -3 dualpkg\m2.py # OK  
c:\code> py -2 dualpkg\m2.py # OK
```

I wreszcie użycie pełnych ścieżek pakietów działa w obu trybach użytkowania, jak i w obu wersjach Pythona, o ile katalog główny pakietu znajduje się w ścieżce wyszukiwania modułów (musi być tam dodany, aby mógł być używany gdzie indziej):

```
# code\dualpkg\m2.py  
import dualpkg.m1 as m1 # oraz: set PYTHONPATH=c:\code  
c:\code> py -3  
>>> import dualpkg.m2 # OK  
C:\code> py -2  
>>> import dualpkg.m2 # OK  
c:\code> py -3 dualpkg\m2.py # OK  
c:\code> py -2 dualpkg\m2.py # OK
```

Podsumowując, o ile nie chcesz i nie możesz izolować modułów w podkatalogach poniżej katalogu skryptów, importowanie pełnych ścieżek pakietów jest prawdopodobnie lepszym rozwiązaniem niż importowanie względne wewnątrz pakietów — chociaż użycie pełnych ścieżek wymaga wpisania większej ilości kodu, jest rozwiązaniem bardziej uniwersalnym, obsługuje wszystkie przypadki zastosowania i działa tak samo zarówno w wersji 2.x, jak i 3.x. Oczywiście istnieją jeszcze inne sztuczki pozwalające na wykonanie takiego zadania, obejmujące dodatkowe czynności (np. ręczne ustawienie ścieżki sys.path w kodzie), ale pominiemy je tutaj, ponieważ są one bardziej zagmatwane i opierają się na wykorzystaniu semantyki importu, która jest podatna na błędy, podczas gdy importowanie z użyciem pełnych ścieżek opiera się tylko na podstawowych mechanizmach pakietów.

Oczywiście zakres, w jakim może to wpływać na Twoje moduły, może się różnić w zależności od pakietu; importy bezwzględne mogą wymagać wprowadzenia zmian do organizacji katalogów pakietów, a z kolei importy względne mogą się zakończyć niepowodzeniem, jeżeli moduł lokalny zostanie przeniesiony w inne miejsce.



Pamiętaj również o zmianach, jakie w tym zakresie mogą pojawić się w przyszłych wersjach języka Python. Chociaż ta książka dotyczy tylko Pythona do wersji 3.3, w dokumentach PEP (ang. *Python Enhancement Proposals*) mówi się o możliwościach rozwiązania niektórych problemów z pakietami w Pythonie 3.4, być może nawet zezwalając na względne importowanie w trybie skryptu. Z drugiej strony zakres i wynik tej inicjatywy jest niepewny i działałby tylko w wersji 3.4 i nowszych; podane tutaj rozwiązanie z pełnymi ścieżkami jest uniwersalne i neutralne dla wersji. Oznacza to, że możesz czekać na zmiany wprowadzane w kolejnych wersjach linii 3.x lub po prostu używać sprawdzonych i zawsze działających pełnych ścieżek pakietów.

Pakiety przestrzeni nazw w Pythonie 3.3

Teraz gdy dowiedziałeś się już wielu rzeczy o pakietach i importowaniu względnym wewnątrz pakietów, warto również zauważać, że istnieje nowa opcja, która modyfikuje niektóre omawiane tutaj wcześniej pomysły. Przynajmniej abstrakcyjnie, od wersji 3.3, Python ma cztery modele importu. Od najstarszego do najnowszego wyglądają one następująco:

Podstawowe, proste importowanie modułów: `import mod, from mod import attr`

Pierwszy, oryginalny model importowania: import plików i ich zawartości względem ścieżki wyszukiwania modułów `sys.path`.

Import pakietów: `import dir1.dir2.mod, from dir1.mod import attr`

Import, który dodaje rozszerzenia ścieżki katalogu względem ścieżki wyszukiwania modułu `sys.path`, gdzie każdy pakiet jest zawarty w jednym katalogu i ma plik inicjujący; dostępne w *Pythonie 2.x i 3.x*.

Import względem pakietu: `from . import mod` (względny), `import mod` (bezwzględny)

Model zastosowany do importu wewnątrz pakietów, omawianego w poprzedniej sekcji, wraz z jego względnymi lub bezwzględnymi schematami wyszukiwania dla importów z użyciem kropek lub bez; dostępny, ale różniący się w wersjach 2.x i 3.x *Pythona*.

Pakietы przestrzeni nazw: `import splitdir.mod`

Nowy model pakietu przestrzeni nazw, który omówimy w tym podrozdziale; pozwala pakietom składać się z wielu katalogów i nie wymaga pliku inicjującego; wprowadzony w *Pythonie 3.3*.

Pierwsze dwa modele są całkowicie samowystarczalne, trzeci zaostrza kolejność wyszukiwania i rozszerza składnię dla importów wewnątrz pakietu, a czwarty całkowicie odwraca niektóre podstawowe koncepcje i wymagania poprzedniego modelu pakietu. W praktyce spowodowało to, że Python 3.3 i nowsze wersje posiadają teraz tylko dwa warianty pakietów:

- Oryginalny model, obecnie znany jako *pakietы regularne* lub po prostu *zwykłe* (ang. *regular packages*).
- Model alternatywny, znany jako *pakietы przestrzeni nazw* (ang. *namespace packages*).

Jest to podobne w założeniu do dychotomii klasycznego i nowego stylu modeli klas, które spotkamy w następnej części tej książki, choć w przypadku pakietów nowy styl jest raczej dodatkiem do starego. Oryginalne i nowe modele pakietów nie wykluczają się wzajemnie i mogą być używane jednocześnie w tym samym programie. W rzeczywistości nowy model pakietów przestrzeni nazw działa jako *opcja rezerwowa*, używana tylko wtedy, gdy normalne moduły i standardowe pakiety o tej samej nazwie nie są obecne w ścieżce wyszukiwania modułów.

Uzasadnienie dla istnienia pakietów przestrzeni nazw jest zakorzenione w celach *instalacji* pakietów, które mogą wydawać się nie do końca oczywiste; jeżeli jesteś odpowiedzialny za takie zadania, powinieneś uważnie zapoznać się z dokumentem PEP dotyczącym tej funkcji. Krótko mówiąc, pakiety przestrzeni nazw radykalnie rozwiązują problem związany z możliwością wystąpienia kolizji wielu plików `__init__.py` podczas scalania części pakietu poprzez całkowite usunięcie tego pliku. Ponadto dzięki zapewnieniu standardowej obsługi pakietów, które można podzielić na wiele katalogów i umieszczać w kolejnych wpisach w ścieżce `sys.path`, pakiety przestrzeni nazw zwiększą elastyczność instalacji i zapewniają zintegrowany mechanizm zastępujący wiele niekompatybilnych ze sobą rozwiązań, które były używane do tej pory.

Chociaż jest jeszcze zbyt wcześnie, aby ocenić rzeczywistą przydatność pakietów przestrzeni nazw, przeciętni użytkownicy Pythona zwykle określają je jako bardzo użyteczne i alternatywne rozszerzenie zwykłego modelu pakietu — takie, które nie wymaga plików inicjujących i pozwala na użycie dowolnego katalogu kodu jako pakietu do zimportowania. Aby przekonać się dlaczego, przejdziemy do szczegółów.

Semantyka pakietów przestrzeni nazw

Pakiet przestrzeni nazw nie różni się zasadniczo od zwykłego pakietu; to po prostu nieco inny sposób tworzenia pakietów. Co więcej, na najwyższym poziomie nadal są one względne w stosunku do ścieżki `sys.path`: pierwszy komponent ścieżki pakietu przestrzeni nazw (zapisywanej z kropkami) musi nadal znajdować się w normalnej ścieżce wyszukiwania modułów.

Jednak pod względem budowy fizycznej oba rodzaje pakietów mogą się znacznie od siebie różnić. Zwykłe pakiety nadal muszą być zapisane w jednym katalogu i posiadać plik `__init__.py`, który jest uruchamiany automatycznie. W przeciwieństwie do nich nowe pakiety przestrzeni nazw nie mogą zawierać pliku `__init__.py` i mogą składać się z wielu katalogów wykorzystywanych podczas importu. W rzeczywistości żaden z katalogów tworzących pakiet przestrzeni nazw nie może mieć pliku `__init__.py`, ale zawartość zagnieżdzona w każdym z nich jest traktowana jak pojedynczy pakiet.

Algorytm importu

Aby naprawdę zrozumieć pakiety przestrzeni nazw, musimy zatrzymać się pod „maską”, aby zobaczyć, jak działa operacja importowania w wersji 3.3. Podczas importu Python nadal iteruje po każdym katalogu w ścieżce wyszukiwania modułów — zdefiniowanej przez `sys.path` dla importów bezwzględnych oraz według lokalizacji pakietu dla importów względnych i komponentów zagnieżdzonych w ścieżkach pakietów — tak jak to się odbywało w Pythonie 3.2 i wersjach wcześniejszych. Jednak szukając zaimportowanego modułu lub pakietu o nazwie np. `spam` w wersji 3.3, dla każdego *katalogu* w ścieżce wyszukiwania modułów Python sprawdza większą liczbę kryteriów w następującej kolejności:

1. Jeżeli zostanie znaleziony plik `nazwa_katalogu\spam__init__.py`, importowany i zwracany jest zwykły pakiet.
2. Jeżeli zostanie znaleziony plik `nazwa_katalogu\spam.py` (lub `spam.pyc` lub inne rozszerzenie modułu), importowany i zwracany jest zwykły pakiet.
3. Jeżeli zostanie znaleziony katalog `nazwa_katalogu\spam`, zostanie on zapisany, a skanowanie będzie kontynuowane w następnym katalogu ze ścieżki wyszukiwania.
4. Jeżeli żaden z powyższych plików lub katalogów nie zostanie znaleziony, skanowanie będzie kontynuowane od następnego katalogu ze ścieżki wyszukiwania.

Jeżeli skanowanie ścieżki wyszukiwania zakończy się bez zwracania modułu lub pakietu według kroków 1. lub 2., a co najmniej jeden katalog został zapisany w kroku 3., wówczas tworzony jest *pakiet przestrzeni nazw*.

Tworzenie pakietu przestrzeni nazw odbywa się natychmiast i nie jest odraczane do momentu wystąpienia importu na poziomie niżej. Nowy pakiet przestrzeni nazw ma atrybut `__path__` ustawiony na iterowalną listę ścieżek katalogów, które zostały znalezione i zarejestrowane podczas skanowania w kroku 3., ale nie posiada atrybutu `__file__`.

Atrybut `__path__` jest następnie wykorzystywany w późniejszych, głębszych dostępach do przeszukiwania wszystkich składników pakietu — każdy katalog w ścieżce `__path__` pakietu przestrzeni nazw jest przeszukiwany, gdy poszukiwane są bardziej zagnieżdzone elementy, podobnie jak to miało miejsce w przypadku katalogu zwykłego pakietu.

Patrząc z innej strony, atrybut `__path__` pakietu przestrzeni nazw pełni tę samą rolę dla komponentów niższego poziomu, co `sys.path` dla elementów ścieżek importu pakietów; staje się „ścieżką nadzczną” pozwalającą na dostęp do położonych niżżej elementów przy użyciu tej samej czteroetapowej procedury, którą omówiliśmy przed chwilą.

W efekcie pakiet przestrzeni nazw jest rodzajem *wirtualnej konkatenacji* katalogów zdefiniowanych za pomocą wpisów w ścieżce wyszukiwania modułów. Jednak po utworzeniu pakietu przestrzeni nazw nie ma funkcjonalnej różnicy między nim a zwykłym pakietem; obsługuje wszystko, czego nauczyliśmy się dla zwykłych pakietów, w tym importowanie względne wewnątrz pakietu.

Wpływ na zwykłe pakiety: opcjonalne pliki `__init__.py`

Jedną z konsekwencji tej nowej procedury importowania jest to, że od wersji 3.3 Pythona pakiety nie muszą już posiadać plików `__init__.py` — gdy pakiet z jednym katalogiem nie ma tego pliku, będzie traktowany jako pakiet przestrzeni nazw z jednym katalogiem i żadne ostrzeżenie nie będzie wyświetlane. Jest to znaczne złagodzenie wcześniejszych zasad, ale zgodne z powszechnymi uwagami użytkowników; wiele pakietów nie wymaga kodu inicjalizacji, a mimo to w takich przypadkach i tak konieczne było utworzenie pustego pliku inicjalizacji — począwszy od wersji 3.3, nie jest to już wymagane.

Jednocześnie oryginalny, standardowy model pakietu jest nadal w pełni obsługiwany i automatycznie uruchamia kod znajdujący się w pliku `__init__.py`, co spełnia rolę *punktu zaczepienia dla inicjalizacji pakietu*. Ponadto gdy wiadomo, że pakiet nigdy nie będzie częścią podzielonego pakietu przestrzeni nazw, kodowanie go jako zwykłego pakietu z użyciem pliku `__init__.py` daje pewną *przewagę wydajności*. Tworzenie i ładowanie zwykłego pakietu następuje natychmiast, gdy zostanie on zlokalizowany w ścieżce wyszukiwania. W przypadku pakietów przestrzeni nazw przed utworzeniem pakietu muszą zostać przeskanowane wszystkie wpisy ze ścieżki wyszukiwania. Bardziej formalnie mówiąc, zwykłe pakiety zatrzymują algorytm z poprzedniej sekcji już na pierwszym kroku, podczas gdy pakiety przestrzeni nazw nie.

Zgodnie z dokumentem PEP opisującym tę zmianę, usuwanie obsługi zwykłych pakietów nie jest planowane — a przynajmniej tak to wygląda na dzień dzisiejszy; w projektach typu open source zmiany są zawsze możliwe (w poprzedniej edycji tej książki wspominaliśmy o planach zmian w metodach formatowania łańcuchów i względnego importu w wersji 2.x, które zostały później porzucone), więc jak zwykle powinieneś uważnie śledzić przyszłe zmiany w tym zakresie. Biorąc pod uwagę przewagę wydajności i możliwość automatycznej inicjalizacji zwykłych pakietów, wydaje się jednak mało prawdopodobne, że ich obsługa zostanie całkowicie usunięta.

Pakiety przestrzeni nazw w akcji

Aby zobaczyć, jak działają pakiety przestrzeni nazw w praktyce, przyjrzyjmy się dwóm następującym modułem i niezbędnej strukturze katalogów — z dwoma podkatalogami o nazwie `sub` zlokalizowanymi w różnych katalogach macierzystych, `dir1` i `dir2`:

```
C:\code\ns\dir1\sub\mod1.py  
C:\code\ns\dir2\sub\mod2.py
```

Jeżeli dodamy zarówno `dir1`, jak i `dir2` do ścieżki wyszukiwania modułów, katalog `sub` staje się pakietem przestrzeni nazw obejmującym oba katalogi, z dwoma plikami modułów dostępnymi pod tą nazwą, pomimo że znajdują się one w osobnych katalogach fizycznych. Oto zawartość plików i wymagane ustawienia ścieżek w systemie Windows: nie ma tutaj plików `__init__.py` — w rzeczywistości *nie może ich być* w pakietach przestrzeni nazw, ponieważ jest to ich najważniejsza różnica fizyczna:

```
c:\code> mkdir ns\dir1\sub # Dwa podkatalogi o tej samej  
nazwie w różnych katalogach  
  
c:\code> mkdir ns\dir2\sub # Podobnie w systemach innych niż  
Windows
```

```
c:\code> type ns\dir1\sub\mod1.py          # Pliki modułów w różnych
katalogach
print(r'dir1\sub\mod1')
c:\code> type ns\dir2\sub\mod2.py
print(r'dir2\sub\mod2')
c:\code> set PYTHONPATH=C:\code\ns\dir1;C:\code\ns\dir2
```

Po bezpośrednim zimportowaniu w wersji 3.3 Pythona i nowszych pakiet przestrzeni nazw staje się *wirtualnym połączeniem* poszczególnych składników katalogu i umożliwia poprzez normalny import dostęp do dalszych zagnieżdżonych części za pomocą jednej, złożonej nazwy:

```
c:\code> C:\Python33\python
>>> import sub
>>> sub                         # Pakiety przestrzeni nazw: zagnieżdżone ścieżki
wyszukiwania
<module 'sub' (namespace)>
>>> sub.__path__
[_NamespacePath(['C:\\\\code\\\\ns\\\\dir1\\\\sub', 'C:\\\\code\\\\ns\\\\dir2\\\\sub'])]
>>> from sub import mod1
dir1\sub\mod1
>>> import sub.mod2             # Zawartość z dwóch różnych katalogów
dir2\sub\mod2
>>> mod1
<module 'sub.mod1' from 'C:\\\\code\\\\ns\\\\dir1\\\\sub\\\\mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\\\code\\\\ns\\\\dir2\\\\sub\\\\mod2.py'>
```

Dzieje się tak również wtedy, gdy *natychmiast* importujemy nazwę pakietu za pośrednictwem przestrzeni nazw — ponieważ pakiet przestrzeni nazw jest tworzony przy pierwszym użyciu, czas przeszukiwania ścieżek jest nieistotny:

```
c:\code> C:\Python33\python
>>> import sub.mod1
dir1\sub\mod1
>>> import sub.mod2             # Jeden pakiet składający się z dwóch
katalogów
dir2\sub\mod2
>>> sub.mod1
<module 'sub.mod1' from 'C:\\\\code\\\\ns\\\\dir1\\\\sub\\\\mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\\\code\\\\ns\\\\dir2\\\\sub\\\\mod2.py'>
```

```
>>> sub
<module 'sub' (namespace)>
>>> sub.__path__
[_NamespacePath(['C:\\\\code\\\\ns\\\\dir1\\\\sub', 'C:\\\\code\\\\ns\\\\dir2\\\\sub'])]
```

Co ciekawe, *import względny* działa również w pakietach przestrzeni nazw — pokazana poniżej instrukcja importu względnego odwołuje się do pliku w pakiecie, mimo że sam plik, do którego się odwołuje, znajduje się w innym katalogu:

```
c:\\code> type ns\\dir1\\sub\\mod1.py
from . import mod2          # próba wykonania "from . import string"
nadal kończy się niepowodzeniem

print(r'dir1\\sub\\mod1')

c:\\code> C:\\Python33\\python
>>> import sub.mod1          # Względny import modułu mod2 z innego
katalogu

dir2\\sub\\mod2

dir1\\sub\\mod1

>>> import sub.mod2          # Wcześniej zaimportowany moduł nie jest
wykonywany ponownie

>>> sub.mod2

<module 'sub.mod2' from 'C:\\\\code\\\\ns\\\\dir2\\\\sub\\\\mod2.py'>
```

Jak widać, pakiety przestrzeni nazw są pod każdym względem takie jak zwykłe pakiety z jednym katalogiem, z wyjątkiem dzielonego magazynu fizycznego — właśnie dlatego pakiety przestrzeni nazw składające się z jednego katalogu bez pliku `__init__.py` są dokładnie takie same jak zwykłe pakiety, z tym że nie posiadają kodu inicjującego, który mógłby być uruchamiany automatycznie.

Zagnieżdżanie pakietów przestrzeni nazw

Pakiety przestrzeni nazw obsługują nawet dowolne *zagnieżdżanie* — po utworzeniu pakiet przestrzeni nazw spełnia na swoim poziomie zasadniczo tę samą rolę, co ścieżka `sys.path` na górze, stając się „ścieżką nadzczną” dla niższych poziomów. Kontynuując przykład z poprzedniej sekcji:

```
c:\\code> mkdir ns\\dir2\\sub\\lower    # Głębiej zagnieżdżone komponenty
c:\\code> type ns\\dir2\\sub\\lower\\mod3.py
print(r'dir2\\sub\\lower\\mod3')

c:\\code> C:\\Python33\\python
>>> import sub.lower.mod3          # Pakiet przestrzeni nazw zagnieżdżony w
pakiecie przestrzeni nazw

dir2\\sub\\lower\\mod3

c:\\code> C:\\Python33\\python
```

```

>>> import sub                      # Ten sam rezultat przy ręcznym
przechodzeniu do kolejnych poziomów

>>> import sub.mod2

dir2\sub\mod2

>>> import sub.lower.mod3

dir2\sub\lower\mod3

>>> sub.lower                      # Jednokatalogowy pakiet przestrzeni nazw

<module 'sub.lower' (namespace)>

>>> sub.lower.__path__

	NamespacePath(['C:\\\\code\\\\ns\\\\dir2\\\\sub\\\\lower'])

```

W powyższym przypadku `sub` jest pakietem przestrzeni nazw podzielonym na dwa katalogi, a `sub.lower` jest jednokatalogowym pakietem przestrzeni nazw zagnieżdżonym w części `sub` fizycznie zlokalizowanej w katalogu `dir2`. `sub.lower` jest także pakietem przestrzeni nazw będącym odpowiednikiem zwykłego pakietu bez pliku `__init__.py`.

Takie zachowanie zagnieżdżania jest prawdziwe niezależnie od tego, czy niższy komponent jest modułem, zwykłym pakietem, czy innym pakietem przestrzeni nazw — jako nowe ścieżki wyszukiwania importów pakiety przestrzeni nazw pozwalają na swobodne zagnieżdżenie w nich wszystkich trzech rodzajów komponentów:

```

c:\code> mkdir ns\dir1\sub\pkg

C:\code> type ns\dir1\sub\pkg\__init__.py
print(r'dir1\sub\pkg\__init__.py')

c:\code> C:\Python33\python

>>> import sub.mod2                  # Zagnieżdżony moduł

dir2\sub\mod2

>>> import sub.pkg                  # Zagnieżdżony zwykły pakiet

dir1\sub\pkg\__init__.py

>>> import sub.lower.mod3          # Zagnieżdżony pakiet przestrzeni
nazw

dir2\sub\lower\mod3

>>> sub                           # Moduły, pakiety zwykłe i
pakiety przestrzeni nazw

<module 'sub' (namespace)>

>>> sub.mod2

<module 'sub.mod2' from 'C:\\\\code\\\\ns\\\\dir2\\\\sub\\\\mod2.py'>

>>> sub.pkg

<module 'sub.pkg' from 'C:\\\\code\\\\ns\\\\dir1\\\\sub\\\\pkg\\\\__init__.py'>

>>> sub.lower

<module 'sub.lower' (namespace)>

```

```
>>> sub.lower.mod3
<module 'sub.lower.mod3' from 'C:\\\\code\\\\ns\\\\dir2\\\\sub\\\\lower\\\\mod3.py'>
```

Aby jeszcze lepiej zrozumieć działanie pakietów, powinieneś uważnie przeanalizować pliki i katalogi z powyższego przykładu. Jak widać, pakiety przestrzeni nazw bezproblemowo integrują się z poprzednimi modelami importów i rozszerzają je o nowe funkcje.

Pliki nadal mają pierwszeństwo przed katalogami

Jak wyjaśnialiśmy już wcześniej, jednym z zadań plików `_init_.py` w zwykłych pakietach jest zadeklarowanie katalogu jako pakietu — taki plik mówi Pythonowi, aby używał całego katalogu, zamiast poszukiwać możliwego pliku o tej samej nazwie w dalszej ścieżce. Pozwala to uniknąć przypadkowego wybrania podkatalogu niezawierającego żadnego kodu, który przypadkowo pojawiłby się gdzieś w ścieżce wyszukiwania modułów przed właściwym modułem o tej samej nazwie.

Ponieważ pakiety przestrzeni nazw nie wymagają tych specjalnych plików, może się wydawać, że neutralizują to zabezzczenie. Tak jednak nie jest — ponieważ opisany wcześniej algorytm przestrzeni nazw kontynuuje skanowanie ścieżki wyszukiwania po znalezieniu katalogu przestrzeni nazw, pliki znalezione później na tej ścieżce nadal mają wyższy priorytet niż wcześniejsze katalogi bez pliku `_init_.py`. Weźmy na przykład następujące katalogi i moduły:

```
c:\\code> mkdir ns2
c:\\code> mkdir ns3
c:\\code> mkdir ns3\\dir
c:\\code> notepad ns3\\dir\\ns2.py
c:\\code> type ns3\\dir\\ns2.py
print(r'ns3\\dir\\ns2.py!')
```

Katalogu `ns2`, pokazanego tutaj, nie można zaimportować w Pythonie 3.2 i wersjach wcześniejszych — nie jest to zwykły pakiet, ponieważ nie ma w nim pliku inicjującego `_init_.py`. Katalog ten można jednak zaimportować w wersji 3.3 — jest to katalog pakietu przestrzeni nazw w bieżącym katalogu roboczym, który zawsze jest *pierwszym* elementem ścieżki wyszukiwania modułów `sys.path`, niezależnie od ustawień zmiennej `PYTHONPATH`:

```
c:\\code> set PYTHONPATH=
c:\\code> py -3.2
>>> import ns2
ImportError: No module named ns2
c:\\code> py -3.3
>>> import ns2
>>> ns2                                # Jednokatalogowy moduł przestrzeni nazw w
                                         bieżącym katalogu roboczym
<module 'ns2' (namespace)>
>>> ns2.__path__
	NamespacePath(['.\\ns2'])
```

Ale zobacz, co się stanie, gdy katalog zawierający plik o tej samej nazwie co katalog przestrzeni nazw zostanie dodany *później* na ścieżce wyszukiwania za pomocą ustawień zmiennej `PYTHONPATH` — zamiast katalogu zostanie użyty plik, ponieważ Python kontynuuje przeszukiwanie kolejnych pozycji ścieżki wyszukiwania modułów po znalezieniu katalogu pakietu przestrzeni nazw. Proces ten jest przerywany dopiero wtedy, gdy znaleziony zostaje pasujący moduł lub zwykły pakiet albo ścieżka wyszukiwania zostaje całkowicie przeskanowana do końca. Pakiety przestrzeni nazw są zwracane tylko wtedy, gdy po drodze nie znaleziono niczego innego:

```
c:\code> set PYTHONPATH=C:\code\ns3\dir
c:\code> py -3.3
>>> import ns2          # Użyje późniejszego pliku modułu, a nie
katalogu o tej samej nazwie!
ns3\dir\ns2.py!
>>> ns2
<module 'ns2' from 'C:\\code\\ns3\\dir\\ns2.py'>
>>> import sys
>>> sys.path[:2]        # Pierwszy element '' reprezentuje bieżący
katalog roboczy
[ '', 'C:\\code\\ns3\\dir' ]
```

W rzeczywistości ustawienie ścieżki w celu dołączenia modułu działa tak samo jak we wcześniejszych wersjach Pythona, nawet jeżeli katalog przestrzeni nazw o tej samej nazwie pojawia się wcześniej na ścieżce; pakiety przestrzeni nazw są używane w wersji 3.3 tylko w przypadkach, które zakończyłyby się wystąpieniem błędów we wcześniejszych wersjach Pythona:

```
c:\code> py -3.2
>>> import ns2
ns3\dir\ns2.py!
>>> ns2
<module 'ns2' from 'C:\code\ns3\dir\ns2.py'>
```

Z tego też powodu *żadne* katalogi w pakiecie przestrzeni nazw nie mogą zawierać pliku `__init__.py`: gdy tylko algorytm importu znajdzie plik o takiej nazwie, natychmiast zwraca pakiet standardowy i kończy przeszukiwanie ścieżki i przestrzeni nazw. Mówiąc bardziej formalnie, algorytm importu wybiera pakiet przestrzeni nazw tylko na końcu skanowania ścieżki wyszukiwania i zatrzymuje się na krokach 1. lub 2., jeżeli wcześniej zostanie znaleziony zwykły plik pakietu lub modułu.

W rezultacie *zarówno* pliki modułów, jak i zwykłe pakiety w dowolnym miejscu ścieżki wyszukiwania modułów mają pierwszeństwo przed katalogami pakietów przestrzeni nazw. W poniższym przykładzie pakiet przestrzeni nazw o nazwie `sub` istnieje jako konkatenacja podkatalogów o takich samych nazwach z katalogu `dir1` i `dir2` na ścieżce wyszukiwania:

```
c:\code> mkdir ns4\dir1\sub
c:\code> mkdir ns4\dir2\sub
c:\code> set PYTHONPATH=c:\code\ns4\dir1;c:\code\ns4\dir2
c:\code> py -3
```

```
>>> import sub
>>> sub
<module 'sub' (namespace)>
>>> sub.__path__
[NamespacePath(['c:\\code\\ns4\\dir1\\sub', 'c:\\code\\ns4\\dir2\\sub'])]
```

Jednak podobnie jak plik modułu, tak i zwykły *pakiet* dodany na końcu ścieżki wyszukiwania ma pierwszeństwo przed katalogami pakietów przestrzeni nazw o takiej samej nazwie — skanowanie ścieżki wyszukiwania zaczyna wstępnie zapisywać pakiet przestrzeni nazw w katalogu *dir1* jak poprzednio, ale porzuca go, gdy zwykły pakiet zostanie wykryty w katalogu *dir2*:

```
c:\\code> notepad ns4\\dir2\\sub\\__init__.py
c:\\code> py -3
>>> import sub # Użyje zwykłego pakietu z końca ścieżki, a
nie katalogu o tej samej nazwie
>>> sub
<module 'sub' from 'c:\\code\\ns4\\dir2\\sub\\__init__.py'>
```

Choć jest to przydatne rozszerzenie, to jednak ze względu na fakt, że pakiety przestrzeni nazw są dostępne tylko dla użytkowników wersji 3.3 Pythona (i nowszych), nie będziemy go tutaj bardziej szczegółowo omawiać; więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji Twojej wersji Pythona. Jeżeli jesteś zainteresowany zagadnieniami związanymi z importowaniem, powinieneś uważnie zapoznać się zwłaszcza z dokumentem PEP tej zmiany, który zawiera szczegółowe uzasadnienie, dodatkowe informacje i bardziej wyczerpujące przykłady zastosowania.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadziliśmy *model importowania pakietów* Pythona — opcjonalną, lecz przydatną metodę jawnego wymienienia części ścieżki katalogów prowadzących do modułów. Importowanie pakietów nadal odbywa się względem katalogu ze ścieżki wyszukiwania modułów, jednak zamiast pozostawić Pythonowi ręczne przejście ścieżki wyszukiwania, skrypt może podać resztę ścieżki do modułu w sposób jednoznaczny.

Jak widzieliśmy, pakiety nie tylko sprawiają, że importowanie w większych systemach jest bardziej zrozumiałe, ale także upraszczają ustawienia ścieżki wyszukiwania (jeżeli wszystkie operacje importowania pomiędzy katalogami odbywają się względem wspólnego katalogu głównego). Pomagają one również usunąć niejednoznaczność w sytuacji, gdy istnieje większa liczba modułów o tej samej nazwie (dodanie nazwy katalogu zawierającego moduł do importu pakietu pomaga odróżnić moduły).

Omówiliśmy również *model importów względnych*, stosowany wyłącznie w pakietach. Jest to sposób wymuszenia importu plików zdefiniowanych w tym samym pakiecie polegający na zastosowaniu wiodących kropek w ścieżce importu w instrukcji `from`. Metoda ta zastępuje stosowaną w starszych wersjach Pythona i bardziej podatną na błędy regułę domyślnego wyszukiwania modułów w bieżącym katalogu pakietu. Na koniec zbadaliśmy *pakiety przestrzeni nazw* Pythona 3.3, które pozwalają, aby pakiet składał się z wielu katalogów fizycznych; jest to w pewnym sensie „ostatnia deska ratunku” podczas wyszukiwania importów,

która całkowicie usunęła wymagania dotyczące plików inicjalizacyjnych znane z poprzedniego modelu importowania.

W kolejnym rozdziale omówimy kilka bardziej zaawansowanych zagadnień związanych z modułami, takich jak użycie atrybutu `__name__` czy importowanie nazw. Jak zawsze jednak zamkniami bieżący rozdział krótkim quizem sprawdzającym wiadomości w nim przedstawione.

Sprawdź swoją wiedzę — quiz

1. Jaki jest cel umieszczania pliku `__init__.py` w katalogu pakietu modułu?
2. W jaki sposób możemy uniknąć powtarzania pełnej ścieżki pakietu za każdym razem, gdy odnosimy się do zawartości pakietu?
3. Które katalogi wymagają, by znajdował się w nich plik `__init__.py`?
4. Kiedy w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from`?
5. Jaka jest różnica między instrukcją `from mypkg import spam` a `from . import spam`?
6. Czym jest pakiet przestrzeni nazw?

Sprawdź swoją wiedzę — odpowiedzi

1. Plik `__init__.py` służy do deklarowania i inicjalizacji zwykłego pakietu modułu. Python automatycznie wykonuje jego kod za pierwszym razem, gdy w procesie importujemy moduł za pośrednictwem katalogu. Przypisane zmienne pliku stają się atrybutami obiektu modułu utworzonego w pamięci i odpowiadającego temu katalogowi. Nie jest on opcjonalny — nie możemy importować modułów za pomocą składni pakietów, jeżeli katalog nie zawiera tego pliku.
2. Aby bezpośrednio skopiować zmienne z pakietu, należy użyć instrukcji `from` lub `rozszerzenia as` w połączeniu z instrukcją `import`, zastępując ścieżkę krótszym synonimem. W obu przypadkach ścieżka wymieniana jest tylko w jednym miejscu, czyli instrukcji `from` bądź `import`.
3. W Pythonie 3.2 i wersjach wcześniejszych każdy katalog wymieniony w instrukcjach `import` oraz `from` musiał zawierać plik `__init__.py`. Pozostałe katalogi, w tym katalog zawierający pierwszy od lewej strony komponent ścieżki pakietu, nie muszą zawierać tego pliku.
4. W przypadku pakietów instrukcji `import` musimy użyć w miejsce `from` jedynie `wtedy`, gdy potrzebujemy uzyskać dostęp do tej samej zmiennej zdefiniowanej w więcej niż jednej ścieżce. Dzięki instrukcji `import` ścieżka sprawia, że referencje stają się unikalne, natomiast instrukcja `from` pozwala na wykorzystywanie tylko jednej wersji danej nazwy zmiennej (o ile do zmiany nazwy nie użyjesz rozszerzenia `as`).
5. W Pythonie 3.x instrukcja `from mypkg import spam` definiuje import *bezwzględny*: pakiet `mypkg` jest wyszukiwany z pominięciem katalogu pakietu, w którym występuje ta instrukcja, to znaczy przeszukiwana jest wyłącznie ścieżka `sys.path`. Instrukcja `from . import spam` definiuje import *względny*: nazwa `spam` jest poszukiwana w odniesieniu do pakietu, w którym występuje ta instrukcja. W Pythonie 2.x import

bezwzględny najpierw przeszukuje katalog pakietu, zanim przejdzie do ścieżki `sys.path`; import względny działa tak, jak to wcześniej opisywaliśmy.

6. Pakiet przestrzeni nazw jest rozszerzeniem modelu importu dostępnym w Pythonie 3.3 i nowszych wersjach; reprezentuje pakiet składający się z jednego lub większej liczby katalogów, które nie zawierają plików `_init__.py`. Gdy Python znajdzie takie katalogi podczas wyszukiwania importu i nie znajdzie wcześniej prostego modułu lub zwykłego pakietu o takiej nazwie, tworzy pakiet przestrzeni nazw, który jest wirtualnym połączeniem wszystkich znalezionych katalogów o nazwie odpowiadającej nazwie żądanego modułu. Dalsze zagnieżdżone komponenty są wyszukiwane we wszystkich katalogach pakietu przestrzeni nazw. W rezultacie powstaje pakiet bardzo podobny do zwykłego pakietu, ale jego zawartość może być podzielona na wiele katalogów.

[1] Składnia z kropkami została wybrana ze względu na swoją neutralność (niezależność od platformy), ale także dlatego, że ścieżki z instrukcją `import` stają się prawdziwymi ścieżkami obiektów zagnieżdżonych. Składnia ta oznacza również, że jeżeli w instrukcjach `import` zapomnisz o pominięciu rozszerzenia `.py`, możesz otrzymać dziwne błędy. Na przykład Python zakłada, że instrukcja `import mod.py` jest operacją importowania ze ścieżką do katalogu, która najpierw załadowuje plik `mod.py`, później spróbuje załadować plik `mod\py.py`, a na końcu zwróci dość mylący komunikat o błędzie: *No module named py* (brak modułu o nazwie `py`). W wersji 3.3 Pythona ten komunikat o błędzie został poprawiony i obecnie brzmi następująco: *No module named 'mod.py'; mod is not a package* (nie ma modułu o nazwie `mod.py`; `mod` nie jest pakietem).

Rozdział 25. Zaawansowane zagadnienia związane z modułami

Niniejszy rozdział kończy tę część książki zbiorem bardziej zaawansowanych zagadnień związanych z modułami — ukrywaniem danych, modułem `_future_`, zmienną `_name_`, zmianami w ścieżce `sys.path`, narzędziami do wyświetlania danych, importowaniem modułów według nazwy czy przeładowywaniem modułów. Oprócz tego znajdziesz się w nim umieszczone pod koniec każdej części książki omówienie pułapek związanych z przedstawionymi tutaj kwestiami, a także ćwiczenia końcowe.

Po drodze utworzymy bardziej rozbudowane niż dotychczas i przydatne narzędzia łączące w sobie funkcje oraz moduły. Podobnie do funkcji, moduły są bardziej wydajne, kiedy ich interfejsy są dobrze zdefiniowane, dlatego niniejszy rozdział zawiera również omówienie koncepcji związanych z projektowaniem modułów; o części z nich wspomnieliśmy w poprzednich rozdziałach.

Pomimo umieszczenia słowa „zaawansowane” w tytule rozdziału znajdziesz tutaj przede wszystkim omówienie szeregu dodatkowych zagadnień związanych z modułami. Ponieważ niektóre z nich są szeroko stosowane (zwłaszcza sztuczka ze zmienną `_name_`), powinieneś im się koniecznie przyjrzeć, zanim przejdziemy do omawiania klas w kolejnej części książki.

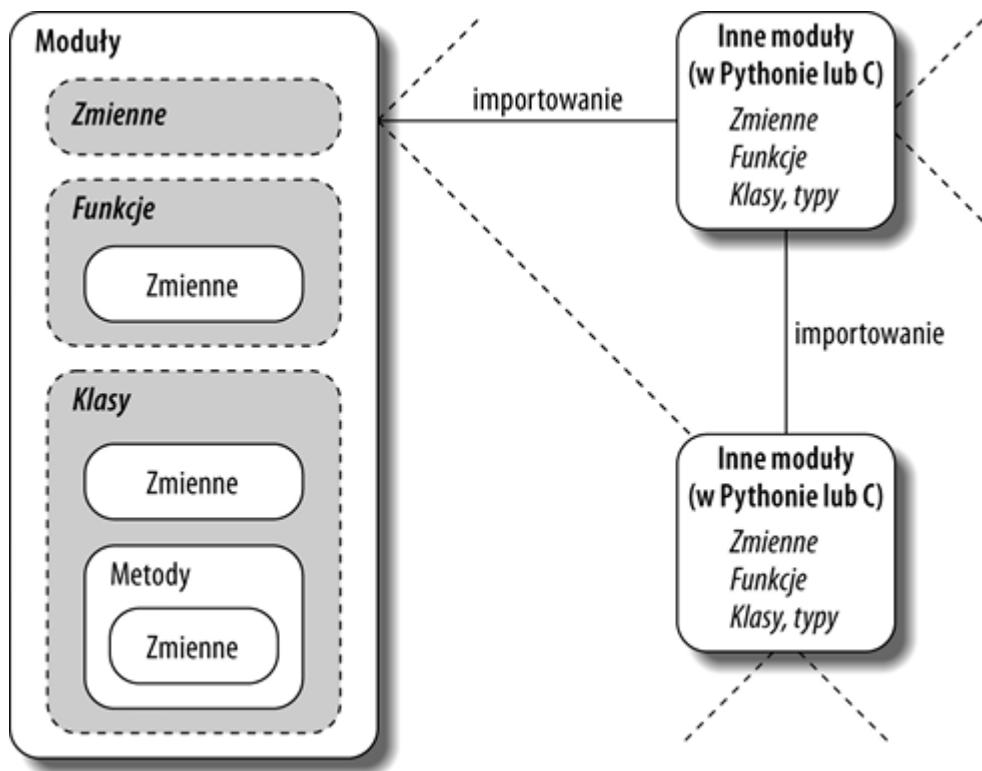
Koncepcje związane z projektowaniem modułów

Podobnie jak w przypadku funkcji, tworzenie modułów nierozerwalnie związane jest z mniejszymi bądź większymi kompromisami projektowymi: musisz zastanowić się, w których modułach umieścić poszczególne funkcje, zaprojektować mechanizmy komunikacji między modułami i tak dalej. Wszystko to stanie się bardziej klarowne, gdy samodzielnie zaczniesz pisać większe programy w języku Python, ale istnieją jednak pewne ogólne reguły, o których warto pamiętać:

- **W Pythonie zawsze jesteś w jakimś module.** Nie da się napisać kodu, który nie jest częścią jakiegoś modułu. Jak wspominaliśmy już pokrótko w rozdziale 17. i rozdziale 21., nawet kod wpisywany w interaktywnej sesji konsoli Pythona w rzeczywistości wchodzi w skład wbudowanego modułu o nazwie `_main_`; jedynymi unikatowymi cechami sesji interaktywnej jest to, że kod jest od razu wykonywany i natychmiast odrzucany, a wyniki działania są automatycznie wyświetlane na ekranie.
- **Minimalizacja sprzężenia modułów: zmienne globalne.** Podobnie jak funkcje, moduły działają najlepiej, jeżeli są napisane jako zamknięte pudełka. Zasadniczo powinny one być tak niezależne od zmiennych globalnych używanych w innych modułach, jak to możliwe, z wyjątkiem importowanych z nich funkcji i klas. Jedyne, co moduł powinien udostępniać światu zewnętrznemu, to używane i definiowane przez niego narzędzia.

- **Maksymalizacja spójności modułów: ujednolicone przeznaczenie.** Staraj się minimalizować liczbę połączeń i zależności między modułami, maksymalizując jednocześnie ich spójność; jeżeli wszystkie komponenty modułu mają wspólne przeznaczenie, mniej prawdopodobna staje się konieczność polegania na zmiennych zewnętrznych.
- **Moduły nie powinny modyfikować zmiennych w innych modułach.** Zilustrowaliśmy to przykładami w rozdziale 17., ale warto tutaj powtórzyć jeszcze raz: używanie zmiennych globalnych zdefiniowanych w innym module jest całkowicie OK (w końcu w taki sposób klient importuje usługi), ale modyfikowanie zmiennych globalnych w innym module jest często objawem problemu na poziomie projektu. Istnieją oczywiście wyjątki od tej reguły, ale powinieneś zawsze próbować przekazywać wyniki za pośrednictwem narzędzi takich jak argumenty funkcji i zwracane wartości, a nie bezpośredniego modyfikowania zmiennych w innych modułach. W przeciwnym razie wartości zmiennych globalnych będą zależały od arbitralnej kolejności operacji przypisania w innych plikach, a same moduły staną się trudniejsze do zrozumienia i ponownego wykorzystania.

Jako swego rodzaju podsumowanie, na rysunku 25.1 przedstawiono środowisko, w którym działają moduły. Moduły zawierają zmienne, funkcje, klasy i inne moduły (jeżeli są importowane). Funkcje mają swoje lokalne zmienne, podobnie jak klasy (są to obiekty znajdujące się wewnątrz modułów, z którymi spotkamy się już w kolejnym rozdziale). Jak mogłeś się przekonać w części IV, funkcje również mogą być zagnieżdżane, ale ostatecznie i tak są zawarte w modułach.



Rysunek 25.1. Środowisko wykonywania modułu. Moduły są importowane, ale same również mogą importować inne moduły napisane w Pythonie lub innym języku, takim jak C, a następnie ich używać. W modułach znajdują się zmienne, funkcje i klasy, za pomocą których moduł realizuje swoje zadania. Funkcje i klasy również mogą zawierać swoje własne zmienne i inne elementy; inaczej mówiąc, programy są po prostu zbiorami modułów

Ukrywanie danych w modułach

Jak mogłeś się sam przekonać, moduł Pythona eksportuje wszystkie zmienne przypisane na najwyższym poziomie jego pliku. Nie ma czegoś takiego, jak deklarowanie, które zmienne powinny być widoczne poza modułem, a które nie. Tak naprawdę nie da się zapobiec modyfikacji zmiennych wewnętrz modułu przez klienta.

W Pythonie ukrywanie danych w modułach jest konwencją, a nie ograniczeniem składniowym. Jeżeli chcemy zniszczyć moduł, usuwając wszystkie jego zmienne, możemy to zrobić, jednak na szczęście nie zdarzyło mi się jeszcze spotkać programisty z takimi zapędami. Niektórzy puryści protestują przeciwko tak liberalnemu stosunkowi do ukrywania danych i twierdzą, że oznacza to, iż Python nie może implementować hermetyzacji (enkapsulacji). Hermetyzacja w Pythonie polega jednak raczej na tworzeniu pakietów, a nie ograniczeń. Rozwiniemy ten pomysł w następnej części książki w odniesieniu do klas, które również nie mają prywatnej składni, ale często mogą naśladować jej efekt w swoim kodzie.

Minimalizacja niebezpieczeństw użycia `from * — _X` oraz `_all_`

W specjalnych przypadkach możemy poprzedzić nazwy zmiennych pojedynczym znakiem `_` (na przykład `_X`), by zapobiec skopiowaniu ich po zimportowaniu zmiennych modułu za pomocą instrukcji `from *`. W zamierzeniach ma to zapobiegać zanieczyszczaniu przestrzeni nazw. Ponieważ instrukcja `from *` kopiuje wszystkie nazwy zmiennych, kod importujący może otrzymać znacznie więcej, niż żądał (w tym zmienne nadpisujące jego własne nazwy). Znaki `_` nie są deklaracjami zmiennych jako „prywatnych” — takie zmienne nadal widzimy i możemy modyfikować za pomocą innych form importowania, takich jak instrukcja `import`.

```
# unders.py

a, _b, c, _d = 1, 2, 3, 4

>>> from unders import *          # Ładuje tylko nazwy, które nie
rozpoczynają się od podkreślenia

>>> a, c
(1, 3)

>>> _b
NameError: name '_b' is not defined

>>> import unders            # Ale inne sposoby importowania
pobierają wszystkie nazwy

>>> unders._b
2
```

Alternatywnie możemy uzyskać efekt ukrycia podobny do konwencji składniowej `_X`, przypisując listę nazw zmiennych do zmiennej `_all_` na najwyższym poziomie modułu. Kiedy korzystamy z tej opcji, instrukcja `from *` skopiuje jedynie zmienne wymienione w liście `_all_`. W rezultacie jest ona przeciwwstewem konwencji zapisu `_X` — lista `_all_` identyfikuje zmienne, które powinny być skopiowane, natomiast konwencja `_X` wskazuje te zmienne, które nie powinny być skopiowane. Python najpierw szuka w module listy `_all_` i kopiuje wszystkie znalezione na niej nazwy, niezależnie od tego, czy rozpoczynają się od znaku podkreślenia. Jeżeli taka lista nie została zdefiniowana, instrukcja `from *` kopiuje wszystkie zmienne, które nie rozpoczynają się od pojedynczego znaku podkreślenia.

```

# alls.py

__all__ = ['a', '_c']                      # __all__ ma pierwszeństwo przed _X
a, b, _c, _d = 1, 2, 3, 4

>>> from alls import *                  # ładuje tylko nazwy z listy
__all__

>>> a, _c
(1, 3)

>>> b
NameError: name 'b' is not defined

>>> from alls import a, b, _c, _d      # Ale inne sposoby importowania
pobierają wszystkie nazwy

>>> a, b, _c, _d
(1, 2, 3, 4)

>>> import alls

>>> alls.a, alls.b, alls._c, alls._d
(1, 2, 3, 4)

```

Podobnie do konwencji zapisu `_X`, lista `__all__` ma znaczenie jedynie dla instrukcji `from *` i nie ma nic wspólnego z deklarowaniem zmiennej jako prywatnej: inne deklaracje importu mogą nadal uzyskiwać dostęp do wszystkich nazw, jak pokazaliśmy w dwóch ostatnich przykładach. Mimo to autorzy modułów mogą wybrać dowolną z tych technik w implementacji modułów, które będą się zachowywać poprawnie podczas importowania za pomocą instrukcji `from *`. W rozdziale 24. przy okazji omawiania list `__all__` w plikach pakietów `__init__.py` możesz się przekonać, że takie listy pozwalają na zadeklarowanie podmodułów, które będą automatycznie ładowane po wykonaniu instrukcji `from *`.

Włączanie opcji z przyszłych wersji Pythona: `__future__`

Zmiany w Pythonie, które potencjalnie mogą uniemożliwić działanie jakiegoś kodu, są zazwyczaj wprowadzane stopniowo. Na początku pojawiają się jako opcjonalne rozszerzenia, które domyślnie są wyłączone. Aby je włączyć, należy skorzystać ze specjalnej postaci instrukcji `import`.

```
from __future__ import nazwa_opcji
```

W przypadku użycia w skrypcie taka instrukcja musi pojawiać się jako pierwsza instrukcja wykonywalna w pliku (zaraz za notką dokumentacyjną i/lub komentarzem), ponieważ umożliwia specjalny sposób komplikacji kodu dla poszczególnych modułów. Można również wstawić tę instrukcję w sesji interaktywnej, by móc eksperymentować z przyszłymi modyfikacjami języka; w takiej sytuacji opcja taka będzie dostępna do końca sesji interaktywnej.

Na przykład w tej książce pokazywaliśmy już, jak używać tej instrukcji w Pythonie 2.x — w rozdziale 5. aktywowaliśmy tak prawdziwe dzielenie z wersji 3.x, w rozdziale 11. korzystaliśmy z nowej wersji funkcji `print`, a w rozdziale 24. używaliśmy mechanizmu bezwzględnego importowania pakietów dostępnego w wersji 3.x. W poprzednich wydaniach tej książki

używaliśmy tego polecenia do zademonstrowania funkcji generatora, co wymagało słowa kluczowego, które nie było jeszcze domyślnie dostępne w ówczesnej wersji Pythona (w tej roli używana była nazwa opcji generatora).

Wszystkie te zmiany mogą potencjalnie doprowadzić do niepoprawnego działania kodu napisanego wcześniej w Pythonie 2.x, więc zostały wprowadzone stopniowo lub oferowane jako opcjonalne rozszerzenia, włączane za pomocą tego specjalnego polecenia. Jednocześnie niektóre nowe zmiany są już teraz dostępne, aby umożliwić Ci napisanie kodu, który będzie zgodny z przyszłymi wersjami Pythona, bo być może będziesz kiedyś do nich dostosowywać swoje obecne programy.

Aby zapoznać się z listą „mechanizmów i opcji z przyszłości”, które możesz już teraz zimportować i włączyć w opisany wyżej sposób, powinieneś uruchomić polecenie `dir` dla modułu `_future_` po jego zimportowaniu (zamiast tego możesz po prostu zapoznać się z dokumentacją biblioteki Pythona). Zgodnie z dokumentacją żadna z zaimplementowanych nazw funkcji nigdy nie zostanie usunięta, więc można bezpiecznie pozostawić polecenie `from _future_ import nnnnn` nawet w kodzie uruchamianym przez wersję Pythona, w której dana funkcja jest normalnie dostępna.

Mieszane tryby użycia — `_name_` oraz `_main_`

Nasza kolejna sztuczka związana z modułami pozwala zarówno zimportować plik jako moduł, jak i uruchomić go jako samodzielny program — jest to powszechnie stosowane w plikach Pythona. Całe rozwiązanie jest tak proste, że niektórzy użytkownicy na początku nie rozumieją jego sedna: każdy moduł posiada wbudowany atrybut o nazwie `_name_`, który Python automatycznie tworzy i przypisuje w następujący sposób:

- Jeżeli plik jest wykonywany jako plik programu najwyższego poziomu, po uruchomieniu atrybut `_name_` ustawiany jest na ciąg znaków `"__main__"`.
- Jeżeli plik jest importowany, atrybut `_name_` jest zamiast tego ustawiany na nazwę modułu, którą znają jego klienci.

W rezultacie moduł może testować swój własny atrybut `_name_` w celu sprawdzenia, czy jest wykonywany, czy też importowany. Założmy na przykład, że tworzymy poniższy plik modułu o nazwie `runme.py`, z którego będziemy eksportować jedną funkcję o nazwie `tester`.

```
def tester():
    print("Jest Gwiazdka w niebie...")
    if __name__ == '__main__':
        tester() # Tylko przy wykonywaniu
                  # A nie przy importowaniu
```

Moduł ten definiuje funkcję, którą inne pliki mogą importować i wykorzystywać w normalny sposób.

```
C:\code> python
>>> import runme
>>> runme.tester()
Jest Gwiazdka w niebie...
```

Jednak na końcu modułu znajduje się również kod, który powoduje wywołanie funkcji `tester`, jeżeli ten moduł zostanie wykonany jako program.

```
C:\code> python runme.py
```

```
Jest Gwiazdka w niebie...
```

W rezultacie atrybut `__name__` modułu służy jako *flaga trybu użycia*, pozwalając na wykorzystanie jego kodu zarówno w postaci biblioteki, którą można zaimportować, jak i skryptu najwyższego poziomu. Choć sztuczka ta jest bardzo prosta, można ją spotkać w prawie każdym prawdziwym pliku programu Pythona, z jakim będziemy mieli do czynienia.

Jednym z najczęściej spotykanych miejsc, w których wykorzystywany jest atrybut `__name__`, jest kod *autotestu* (ang. *self test code*). Mówiąc w skrócie, można umieścić kod testujący eksportowanie modułu w samym module, opakowując go w test atrybutu `__name__` umieszczony w końcowej części pliku. W ten sposób możemy używać tego modułu, *importując* go w klientach, ale także możemy przetestować jego działanie, *uruchamiając* go bezpośrednio z poziomu powłoki systemowej lub w inny sposób.

W praktyce kod autotestu na dole pliku, wykorzystujący atrybut `__name__`, jest chyba najprostszym i najczęściej spotykanym protokołem testów jednostkowych (ang. *unit testing*) w Pythonie. Jest to znacznie wygodniejsze rozwiązanie niż wpisywanie wszystkich testów z poziomu sesji interaktywnej. W rozdziale 36. omówimy inne, często wykorzystywane opcje testowania kodu w Pythonie — jak się przekonasz, moduły `unittest` oraz `doctest` z biblioteki standardowej udostępniają bardziej zaawansowane narzędzia testujące.

Sztuczka z atrybutem `__name__` jest również często wykorzystywana, kiedy piszemy pliki, które można wykorzystywać zarówno jako narzędzia wiersza poleceń, jak i biblioteki narzędzi. Założymy na przykład, że piszemy w Pythonie skrypt odnajdujący pliki. Większe korzyści z kodu odniesiemy, kiedy spakujemy go w funkcję i dodamy na dole pliku test `__name__` automatycznie wywołujący te funkcje, gdy plik wykonywany jest samodzielnie. W ten sposób kod skryptu można wykorzystać również w innych programach.

Testy jednostkowe z wykorzystaniem atrybutu `__name__`

Tak naprawdę już wcześniej w książce spotkaliśmy się ze świetnym przykładem sytuacji, w której sprawdzanie atrybutu `__name__` mogłoby być przydatne. W poświęconym argumentom w rozdziale 18. pokazaliśmy skrypt obliczający wartość minimalną przekazanego zbioru argumentów (plik `minmax.py` w podrozdziale „Przykład z funkcją obliczającą minimum”):

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Kod autotestu
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

Skrypt ten zawiera na dole kod autotestu, zatem możemy go sprawdzać bez konieczności ponownego wpisywania wszystkich poleceń w sesji interaktywnej za każdym razem, gdy go wykonujemy. Problem z jego obecnym zapisem polega na tym, że wyniki działania kodu autotestu będą wyświetlane za każdym razem, gdy moduł ten zostanie zimportowany przez inny plik, który chce go wykorzystać jako narzędzie — nie do końca jest to opcja przyjazna dla użytkownika. Aby to poprawić, musimy opakować kod autotestu w sprawdzenie atrybutu `__name__`, tak by był on uruchamiany tylko wtedy, gdy plik wykonywany jest jako skrypt najwyższego poziomu, a nie zimportowany moduł (nowa, zmodyfikowana wersja tego modułu została zapisana pod nazwą `minmax2.py`):

```
print('Jestem:', __name__)

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y
if __name__ == '__main__':
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Kod autotestu
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

Wyświetlamy tutaj również wartość atrybutu `__name__` na górze pliku, aby ułatwić jej śledzenie. Python tworzy i przypisuje tę zmienną wskazującą tryb użycia zaraz po rozpoczęciu ładowania pliku. Kiedy wykonujemy ten plik jako skrypt najwyższego poziomu, wartość tego atrybutu ustawiana jest na `__main__`, dzięki czemu kod autotestu jest uruchamiany automatycznie.

```
C:\code> python min.py
Jestem: __main__
1
6
```

Jeżeli jednak importujemy plik, wartość atrybutu jest inna niż `__main__`, więc aby wykonać funkcję, musimy w jawnym sposób ją wywołać.

```
C:\code> python
>>> import minmax2
Jestem: minmax2
>>> minmax2.minmax(minmax2.lessthan, 'm', 'i', 'e', 'l', 'o', 'n', 'k', 'a')
'a'
```

I znów, bez względu na to, czy technika ta wykorzystana zostanie do testowania, rezultat będzie taki, że nasz kod możemy wykorzystywać w dwóch różnych rolach — jako moduł biblioteki narzędzi lub jako program wykonywalny.

Zgodnie z tym, co w rozdziale 24. mówiliśmy o importowaniu względnym



pakietów, technika opisana w tej sekcji może również mieć pewne implikacje dla importów wykonywanych przez pliki używane również jako komponenty pakietów w wersji 3.x; nie zmienia to jednak faktu, że nadal można je wykorzystywać przy użyciu importowania bezwzględnego z podaniem pełnej ścieżki pakietu i innych technik. Więcej szczegółów można znaleźć w omówieniu i przykładach z poprzedniego rozdziału.

Przykład — kod działający w dwóch trybach

Poniżej zamieszczamy bardziej rozbudowany przykład modułu, ilustrujący inną, często stosowaną sztuczkę z użyciem atrybutu `_name_`. Moduł `formats.py` definiuje narzędzia formatującełańcuchy znaków na potrzeby kodu importującego, jednak sprawdza także swoją nazwę w celu sprawdzenia, czy wykonywany jest jako skrypt najwyższego poziomu. Jeżeli tak jest, testuje i wykorzystuje argumenty podane w systemowym wierszu poleceń w celu wykonania gotowego lub przekazanego testu. W Pythonie lista `sys.argv` zawiera *argumenty wywołania przekazane z wiersza poleceń* — to lista łańcuchów znaków reprezentujących słowa wpisane w wierszu wywołania polecenia, gdzie pierwszy element jest zawsze nazwą wykonywanego skryptu. Używaliśmy już tego rozwiązania w rozdziale 21. do przekazywania opcji dla narzędzia mierzącego wydajność kodu, ale w bieżącym kontekście chcemy pokazać użyteczność takiego sposobu jako ogólnego mechanizmu przekazywania danych wejściowych:

```
#!python
"""
Plik: formats.py (dla wersji 2.x i 3.x)

Różne wyspecjalizowane narzędzia służące do formatowania
wyświetlanych łańcuchów znaków. Przetestuj mnie za pomocą
gotowego autotestu lub argumentów wiersza poleceń.

Do zrobienia: dodać nawiasy dla liczb ujemnych i inne funkcje
"""

def commas(N):
    """
    Formatowanie dodatniej liczby całkowitej N tak,
    aby była wyświetlana z przecinkami pomiędzy grupami cyfr,
    np. "xxx,yyy,zzz".
    """

    digits = str(N)
    assert digits.isdigit()
    result = ''
    while digits:
        digits, last3 = digits[:-3], digits[-3:]
        result = last3 + ',' + result

    return result
```

```

        result = (last3 + ',' + result) if result else last3
    return result

def money(N, numwidth=0, currency='$'):
    """
    Formatowanie liczby N tak, aby była wyświetlana jako kwota
    w dolarach, z przecinkami, dwoma miejscami dziesiętnymi,
    wiodącym symbolem waluty $ i znakiem, a także opcjonalnym
    dopełnieniem: "$ -xxx,yyy.zz".
    numwidth=0, aby nie dodawać dopełnienia spacjami;
    currency='', aby pominąć symbol waluty, znaki z zestawu
    poza ASCII dla innych symboli walut (np. funt=u'\xA3' lub u'\u00A3').
    """
    sign = '-' if N < 0 else ''
    N = abs(N)
    whole = commas(int(N))
    fract = ('%.2f' % N)[-2:]
    number = '%s%s.%s' % (sign, whole, fract)
    return '%s%s' % (currency, numwidth, number)

if __name__ == '__main__':
    def selftest():
        tests = 0, 1                      # Nie działa: -1, 1.23
        tests += 12, 123, 1234, 12345, 123456, 1234567
        tests += 2 ** 32, 2 ** 100
        for test in tests:
            print(commas(test))
        print('')
        tests = 0, 1, -1, 1.23, 1., 1.2, 3.14159
        tests += 12.34, 12.344, 12.345, 12.346
        tests += 2 ** 32, (2 ** 32 + .2345)
        tests += 1.2345, 1.2, 0.2345
        tests += -1.2345, -1.2, -0.2345
        tests += -(2 ** 32), -(2**32 + .2345)
        tests += (2 ** 100), -(2 ** 100)
        for test in tests:

```

```
    print('%s [%s]' % (money(test, 17), test))

import sys
if len(sys.argv) == 1:
    selftest()
else:
    print(money(float(sys.argv[1]), int(sys.argv[2])))
```

Powyższy plik działa tak samo zarówno w Pythonie 2.x, jak i w wersji 3.x. Po wykonaniu w sposób bezpośredni sprawdza samego siebie jak poprzednio, tym razem jednak wykorzystując opcje z wiersza poleceń w celu kontrolowania działania testu. Aby przekonać się, co wyświetla kod autotestu, wystarczy bezpośrednio uruchomić powyższy plik bez podawania żadnych argumentów w wierszu poleceń — wyniki działania autotestu są zbyt obszerne, aby zamieszczać je tutaj w całości:

```
c:\code> python formats.py

0
1
12
123
1,234
12,345
123,456
1,234,567
...itd...
```

Aby przetestować wybrane wartości, przekaż je w wierszu polecenia wraz z minimalną, żądaną szerokością pola; kod `__main__` skryptu przekaże je do funkcji `money`, która z kolei wywołuje funkcję `commas`:

```
C:\code> python formats.py 999999999 0
$999,999,999.00
C:\code> python formats.py -999999999 0
$-999,999,999.00
C:\code> python formats.py 123456789012345 0
$123,456,789,012,345.00
C:\code> python formats.py -123456789012345 25
$ -123,456,789,012,345.00
C:\code> python formats.py 123.456 0
$123.46
C:\code> python formats.py -123.454 0
$-123.45
```

Tak jak wcześniej, ponieważ kod ten jest przystosowany do podwójnego trybu użycia, możemy normalnie zimportować jego narzędzia w skryptach, innych modułach i sesjach interaktywnych:

```
>>> from formats import money, commas  
>>> money(123.456)  
'$123.46'  
>>> money(-9999999.99, 15)  
'$ -9,999,999.99'  
>>> X = 99999999999999999999  
>>> '%s (%s)' % (commas(X), X)  
'99,999,999,999,999,999,999 (99999999999999999999)'
```

Argumenty wywołania podawane z poziomu wiersza poleceń można wykorzystywać w podobny sposób do przekazywania danych wejściowych do skryptów, które mogą zawierać kod funkcji i klas przystosowany tak, aby mógł on również być wykorzystywany poprzez importowanie w innych modułach. Więcej szczegółowych informacji na temat przetwarzania argumentów wywołania przekazywanych z poziomu wiersza poleceń znajdziesz w sekcji „Argumenty Pythona przekazywane z wiersza poleceń” oraz w dokumentacji modułów getopt, optparse i argparse ze standardowej biblioteki Pythona. W pewnych sytuacjach można także skorzystać z wbudowanej funkcji input, której używaliśmy w rozdziale 3. i w rozdziale 10. do pobierania danych bezpośrednio od użytkownika powłoki zamiast przekazywania ich w wierszu poleceń. Więcej szczegółowych informacji na temat użytej w przykładzie instrukcji assert znajdziesz w rozdziale 34.

	Powinieneś zapoznać się również z zamieszczonym w rozdziale 7. omówieniem nowej składni metody formatowania łańcuchów znaków <code>{,d}</code> , która została udostępniona w Pythonie 2.7, 3.1 i nowszych wersjach. To rozszerzenie formatowania separuje tysiące za pomocą przecinków w sposób podobny do pokazanego w powyższym przykładzie. Przedstawiony tutaj moduł dodaje jednak formatowanie symbolu waluty i służy jako alternatywa dla ręcznego wstawiania przecinka we wcześniejszych wersjach Pythona.
---	--

Symbole walut: Unicode w akcji

Domyślnie funkcja `money` tego modułu używa symbolu dolara, ale obsługuje również inne symbole walut, umożliwiając przekazywanie znaków Unicode spoza zestawu ASCII. Na przykład znak Unicode o kodzie `00A3(hex)` to symbol brytyjskiego funta, a `00A5(hex)` to japoński jen. Symbole walut możesz kodować na różne sposoby, takie jak:

- Zdekodowana wartość znaku Unicode (liczba całkowita) w postaci *ciągu znaków*, ze znakami ucieczki Unicode lub w zapisie heksadecymalnym (dla zachowania kompatybilności z wersją 2.x w takich literałach ciągów w Pythonie 3.3 powinieneś użyć wiodącego znaku `u`).
- Surowa postać znaku zakodowana w postaci *ciągu bajtów*, który jest dekodowany przed przekazaniem, z użyciem szesnastkowych znaków ucieczki (dla zachowania kompatybilności z wersją 3.x w takich literałach ciągów w Pythonie 2.x powinieneś użyć wiodącego znaku `b`).
- Rzeczywisty znak reprezentujący symbol waluty, umieszczony w tekście programu wraz z deklaracją rodzaju strony kodowej, umieszczoną w kodzie źródłowym.

Znaki Unicode omówiliśmy pokrótce w rozdziale 4. i bardziej szczegółowo wróćmy do nich w rozdziale 37., ale ich podstawowe wymagania są dość proste i nasz program jest całkiem dobrym przykładem ich zastosowania. Aby przetestować alternatywne waluty, wpiszemy następujące polecenie w pliku *format_currency.py*, ponieważ w przeciwnym wypadku konieczne byłoby wprowadzenie zbyt wielu poleceń z poziomu sesji interaktywnej:

```
from __future__ import print_function # 2.x
from formats import money
X = 54321.987
print(money(X), money(X, 0, ''))
print(money(X, currency=u'\xA3'), money(X, currency=u'\u00A5'))
print(money(X, currency=b'\xA3'.decode('latin-1')))
print(money(X, currency=u'\u20AC'), money(X, 0, b'\xA4'.decode('iso-8859-15')))
print(money(X, currency=b'\xA4'.decode('latin-1')))
```

Poniżej przedstawiono wyniki działania tego pliku w Pythonie 3.3 w środowisku IDLE oraz w innych, poprawnie skonfigurowanych kontekstach. Działa to tak samo w wersji 2.x, ponieważ wyświetlanie i kodowanie łańcuchów znaków jest przenośne. Zgodnie z rozdziałem 11. import `__future__` umożliwia w wersji 2.x wywołania funkcji `print` z wersji 3.x, a jak wspominaliśmy w rozdziale 4., literały bajtowe `b'...'` z wersji 3.x są traktowane jako proste ciągi znaków w wersji 2.x, a literały Unicode `u'...'` z wersji 2.x są traktowane jak normalne ciągi znaków w wersji 3.3 i nowszych.

```
$54,321.99 54,321.99
£54,321.99 ¥54,321.99
£54,321.99
€54,321.99 €54,321.99
¤54,321.99
```

Jeżeli zadziałało to poprawnie na Twoim komputerze, prawdopodobnie możesz pominąć kilka następnych akapitów. Jednak w zależności od ustawień interfejsu i systemu uruchomienie tego programu i poprawne wyświetlenie symboli walut może wymagać wykonania kilku dodatkowych kroków. Na moim komputerze działa to poprawnie, gdy Python wyświetla wyniki w oknie, próba wyświetlania symboli euro i walut w dwóch ostatnich wierszach kończy się błędami w wierszu polecenia w systemie Windows.

W szczególności nasz skrypt testowy zawsze działa i generuje poprawne wyniki w interfejsie graficznego środowiska IDLE zarówno w wersji 3.x, jak i 2.x, ponieważ kodowanie Unicode jest tam obsługiwane prawidłowo. Poprawne wyniki otrzymasz również w wersji 3.x w systemie Windows, jeżeli przekierujesz dane wyjściowe do pliku i otworzysz je za pomocą Notatnika, ponieważ wersja 3.x koduje wyniki działania w domyślnej formacie Windows:

```
c:\code> formats_currency.py > temp
c:\code> notepad temp
```

Nie działa to jednak w wersji 2.x, ponieważ w tej wersji Python domyślnie próbuje zakodować wyświetlany tekst jako ASCII. Aby wyświetlić wszystkie znaki spoza zestawu ASCII bezpośrednio w oknie wiersza polecenia systemu Windows, na niektórych komputerach może być konieczna zmiana strony kodowej Windows (używanej do renderowania znaków), a także zmiennej środowiskowej `PYTHONIOENCODING` w Pythonie (używanej do określenia

standardowego kodowania strumieni tekstu, w tym tłumaczenia znaków na bajty podczas wyświetlanego na wspólny format Unicode, taki jak UTF-8:

```
c:\code> chcp 65001                      # Konsola ustawiona na stronę kodową Pythona  
c:\code> set PYTHONIOENCODING=utf-8        # Python ustawiony na stronę kodową konsoli  
c:\code> formats_currency.py > temp       # Zarówno wersja 3.x, jak i 2.x zapisuje tekst w formacie UTF-8  
c:\code> type temp                         # Wyniki są poprawnie wyświetlone w konsoli  
c:\code> notepad temp                      # Notatnik również poprawnie rozpoznaje format UTF-8
```

Na niektórych platformach wykonanie tych polecen może nie być konieczne (dotyczy to nawet niektórych wersji systemu Windows). Zrobiłem to, ponieważ mój laptop jest ustawiony na stronę kodową 437 (zestaw znaków US), ale strona kodowa na Twoim komputerze może być inna.

Co ciekawe, jedynym powodem, dla którego ten test działa w Pythonie 2.x, jest to, że wersja 2.x pozwala na *mieszanie* ciągów normalnych i Unicode, o ile zwykły ciąg znaków składa się z 7-bitowych znaków ASCII. W Pythonie 3.3 literał Unicode u'...' z wersji 2.x jest obsługiwany dla zachowania kompatybilności, ale jest traktowany tak samo jak normalne ciągi znaków '...', które są zawsze w formacie Unicode (usunięcie wiodącego znaku u powoduje, że test działa także w wersjach od 3.0 do 3.2, ale psuje kompatybilność z wersją 2.x):

```
c:\code> py -2  
>>> print(u'\xA5' + '1', '%s2' % u'\u00A3')      # 2.x: mieszanka unicode/str dla ciągów ASCII  
¥1 £2  
c:\code> py -3  
>>> print(u'\xA5' + '1', '%s2' % u'\u00A3')      # 3.x: ciąg jest Unicode, u'' jest opcjonalne  
¥1 £2  
>>> print('\xA5' + '1', '%s2' % '\u00A3')  
¥1 £2
```

Jak już wspominaliśmy, znacznie więcej informacji na temat Unicode znajdziesz w rozdziale 37. — jest to temat, który wielu użytkowników uważa za nieco peryferyjny, ale który może pojawić się nawet w stosunkowo prostych zastosowaniach, takich jak nasz przykład! Najważniejszym wnioskiem jest to (jeśli nie liczyć kwestii operacyjnych), że starannie przygotowany skrypt często potrafi obsługiwać Unicode zarówno w wersji 3.x, jak i 2.x.

Notki dokumentacyjne: dokumentacja modułu w działaniu

Wreszcie, ponieważ główny plik tego przykładu korzysta z *notek dokumentacyjnych*, wprowadzonych w rozdziale 15., możemy również użyć funkcji help lub trybu przeglądarki pakietu PyDoc do wyświetlenia opisu narzędzi — moduły są niemal automatycznie narzędziami

ogólnego zastosowania. Poniżej pokazujemy funkcję `help` w działaniu, a na rysunku 25.2 przedstawiamy notkę dokumentacyjną wyświetlzoną w systemie PyDoc.

```
>>> import formats
>>> help(formats)
Help on module formats:

NAME
    formats

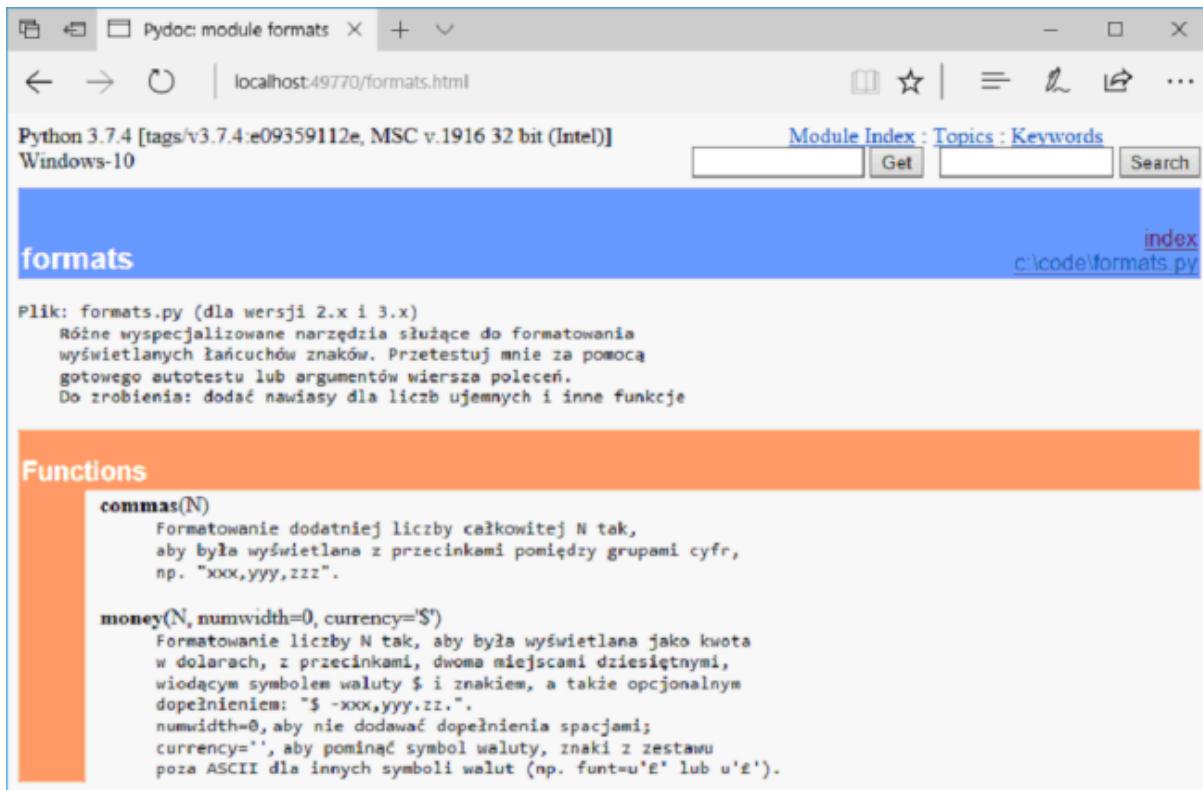
DESCRIPTION
    Plik: formats.py (dla wersji 2.x i 3.x)

    Różne wyspecjalizowane narzędzia służące do formatowania
    wyświetlanych łańcuchów znaków. Przetestuj mnie za pomocą
    gotowego autotestu lub argumentów wiersza poleceń.

    Do zrobienia: dodać nawiasy dla liczb ujemnych i inne funkcje

FUNCTIONS
    commas(N)
        Formatowanie dodatniej liczby całkowitej N tak,
        aby była wyświetlana z przecinkami pomiędzy grupami cyfr,
        np. "xxx,yyy,zzz".
    money(N, numwidth=0, currency='$')
        Formatowanie liczby N tak, aby była wyświetlana jako kwota
        w dolarach, z przecinkami, dwoma miejscami dziesiętnymi,
        wiodącym symbolem waluty $ i znakiem, a także opcjonalnym
        dopełnieniem: "$ -xxx,yyy.zz.".
        numwidth=0, aby nie dodawać dopełnienia spacjami;
        currency='', aby pominąć symbol waluty, znaki z zestawu
        poza ASCII dla innych symboli walut (np. funt=u'\xA3' lub u'\u00A3').

FILE
    c:\code\formats.py
```



Rysunek 25.2. Plik format.py wyświetlony w systemie PyDoc, uzyskany przez uruchomienie polecenia py -3 -m pydoc -b (w wersji 3.2 i nowszych) oraz kliknięcie pliku w indeksie (patrz rozdział 15.)

Modyfikacja ścieżki wyszukiwania modułów

Czas powrócić do omawiania bardziej ogólnych tematów związanych z modułami. W rozdziale 22. dowiedziałeś się, że ścieżka wyszukiwania modułów jest listą katalogów, którą można dostosować do własnych potrzeb za pomocą zmiennej środowiskowej PYTHONPATH, a także plików z rozszerzeniem .pth. Do tej pory nie pokazałem jednak, w jaki sposób program napisany w Pythonie może sam zmodyfikować ścieżkę wyszukiwania, zmieniając wbudowaną listę sys.path. Tak jak pokazywaliśmy w rozdziale 22., lista sys.path inicjalizowana jest po uruchomieniu, ale później możemy jej komponenty usuwać, dodawać i ustawać od nowa w dowolny sposób.

```
>>> import sys  
>>> sys.path  
[ '', 'C:\\\\temp', 'C:\\Windows\\\\system32\\\\python33.zip', ...pozostałe  
usunięto...]  
>>> sys.path.append('C:\\\\sourcedir') # Rozszerzenie ścieżki  
wyszukiwania modułów
```

```
>>> import string # Wszystkie importy przeszukują
nowy katalog na końcu
```

Po dokonaniu takiej zmiany będzie ona miała wpływ na wszystkie przyszłe operacje importowania w Pythonie, ponieważ wszystkie operacje importowania i wszystkie pliki współdzielą jedną listę `sys.path` (podczas uruchamiania programu w pamięci jest tylko jedna kopia danego modułu — właśnie dlatego istnieje polecenie `reload`). W praktyce możemy tę listę zmieniać w dowolny sposób:

```
>>> sys.path = [r'd:\\temp'] # Zmiana ścieżki wyszukiwania
modułów

>>> sys.path.append('c:\\\\lp5e\\\\examples') # Tylko dla tego procesu

>>> sys.path.insert(0, ...)

>>> sys.path
['..', 'd:\\\\temp', 'c:\\\\lp5e\\\\examples']

>>> import string

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'string'
```

Tym samym pokazaną technikę można wykorzystać do dynamicznej konfiguracji ścieżki wyszukiwania wewnętrz programu napisanego w Pythonie. Należy jednak pamiętać — jeżeli usuniemy ze ścieżki kluczowy katalog, możemy utracić dostęp do kluczowych narzędzi. W poprzednim przykładzie nie mieliśmy już na przykład dostępu do modułu `string`, ponieważ usunęliśmy katalog biblioteki źródłowej Pythona ze ścieżki wyszukiwania!

Należy również pamiętać, że takie ustawienia ścieżki `sys.path` trwają jedynie na czas działania sesji Pythona lub programu (z technicznego punktu widzenia: *procesu*), który je utworzył. Nie są one zachowywane po zakończeniu działania Pythona. Konfiguracja ścieżki ze zmiennej środowiskowej `PYTHONPATH` i plików `.pth` zachowywana jest w systemie operacyjnym, a nie w działającym programie utworzonym w Pythonie, przez co jest bardziej globalna: jest pobierana przez każdy program działający na komputerze i zachowywana również po jego zakończeniu. W niektórych systemach pierwszy z pokazanych sposobów może być ustawiany osobno dla poszczególnych użytkowników, a drugi ma szersze zastosowanie dla wszystkich instalacji.

Rozszerzenie `as` dla instrukcji `import` oraz `from`

Instrukcje `import` oraz `from` zostały rozszerzone w taki sposób, aby pozwalać na nadanie zaimportowanym zmiennym innych nazw w naszym skrypcie. Używaliśmy już tego rozszerzenia wcześniej, ale teraz pokażemy kilka dodatkowych szczegółów. Na przykład poniższa instrukcja `import`:

```
import nazwamodułu as nazwa
```

jest odpowiednikiem następującego kodu, który zmienia nazwę modułu tylko w zasięgu importera (dla innych plików moduł ten jest nadal dostępny pod oryginalną nazwą):

```
import nazwamodułu
```

```
nazwa = nazwamodułu
del nazwamodułu                                # Nie zachowujemy oryginalnej
nazwy
```

Po takiej operacji importowania możemy (i tak naprawdę musimy) używać nazwy wymienionej po `as` w każdym odwołaniu do modułu. Działa to również dla instrukcji `from`, przypisując zmienną zimportowaną z pliku do innej zmiennej w zasięgu importera; tak jak w poprzednim przykładzie otrzymujesz tutaj tylko nową nazwę i nie zachowujesz nazwy oryginalnej.

```
from nazwamodułu import nazwaatrybutu as nazwa      # od tej chwili
używamy nowej nazwy
```

Jak wspominaliśmy w rozdziale 23., rozszerzenie to jest często wykorzystywane do tworzenia krótszych synonimów długich nazw, a także w celu uniknięcia konfliktów nazw, kiedy używasz w skrypcie nazwy, która w przeciwnym razie zostałaby nadpisana za pomocą normalnej instrukcji `import`.

```
import naprawdędługanazwamodułu as nazwa          # Użycie krótszego skrótu
name.func()

from moduł1 import narzędzie as narz1            # "narzędzie" może istnieć
tylko raz

from moduł2 import narzędzie as narz2

narz1(); narz2()
```

Przydaje się to również w przypadku podawania krótkich, prostych nazw całych ścieżek do katalogów i unikania kolizji nazw podczas importowania pakietów, które opisywaliśmy w rozdziale 24.

```
import dir1.dir2.mod as mod                      # Pełna ścieżka podana tylko
raz

mod.func()

from dir1.dir2.mod import func as modfunc        # Zmieniamy nazwę na
unikatową, jeżeli to konieczne

modfunc()
```

Jest to również coś w rodzaju zabezpieczenia przed zmianami nazw: jeżeli nowa wersja biblioteki zmieni nazwę modułu lub narzędzia, z których korzysta Twój kod w szerokim zakresie, lub zapewni nową alternatywę, której wolisz użyć, możesz po prostu zmienić jej nazwę na poprzednią podczas importowania i uniknąć w ten sposób niepoprawnego działania kodu:

```
import nowa_nazwa as stara_nazwa
from library import nowa_nazwa as stara_nazwa
...i możesz nadal używać starej nazwy aż do momentu, kiedy będziesz miał
czas, aby zaktualizować cały kod...
```

Takie podejście może na przykład rozwiązać problem niektórych zmian w bibliotece 3.x (np. moduł `tkinter` z wersji 3.x kontra moduł `Tkinter` z wersji 2.x), choć często takie zmiany to znacznie więcej niż tylko nowa nazwa!

Przykład — moduły są obiektami

Ponieważ moduły udostępniają większość swoich najciekawszych właściwości w postaci wbudowanych atrybutów, łatwo jest pisać programy zarządzające innymi programami. Zazwyczaj nazywamy je *metaprogramami*, ponieważ działają one ponad innymi systemami. Nazywa się to również *introspekcją*, ponieważ programy mogą widzieć i przetwarzac mechanizmy wewnętrzne obiektów. Introspekcja jest koncepcją dosyć zaawansowaną, jednak może być bardzo użyteczna do tworzenia narzędzi programistycznych.

Na przykład, aby otrzymać atrybut o nazwie `name` z modułu o nazwie `M`, możemy skorzystać z kwalifikacji (składni z kropką) lub zindeksować słownik atrybutów modułu udostępniany jako wbudowany atrybut `__dict__` — spotkaliśmy się z nim w rozdziale 23. Python eksportuje również listę wszystkich załadowanych modułów jako słownik `sys.modules` (czyli atrybut `modules` modułu `sys`) i udostępnia wbudowaną funkcję `getattr` pozwalającą na pobranie atrybutów z ich łańcuchów nazw. Przypomina to kod `object.attr`, jednak `attr` jest wyrażeniem zwracającym łańcuch znaków w czasie wykonania. Z tego powodu poniższe wyrażenia pozwalają uzyskać dostęp do tego samego atrybutu i obiektu:^[1]

```
M.name                                # Kwalifikacja obiektu poprzez
  atrybut

M.__dict__['name']                      # Ręczne indeksowanie słownika
  przestrzeni nazw

sys.modules['M'].name                   # Ręczne indeksowanie tabeli
  załadowanych modułów

getattr(M, 'name')                     # Wywołanie wbudowanej funkcji
  pobierającej
```

Udostępniając w ten sposób mechanizmy wewnętrzne modułu, Python pomaga nam tworzyć programy o programach. Poniżej znajduje się na przykład moduł `mydir.py`, wprowadzający te koncepcje w życie w celu zaimplementowania własnej wersji wbudowanej funkcji `dir`. Definiuje on i eksportuje funkcję o nazwie `listing`, przyjmującą obiekt modułu jako argument i wyświetlającą sformatowany i posortowany listing przestrzeni nazw modułu:

```
#!/python
"""

mydir.py: moduł wyświetlający przestrzenie nazw innych modułów

"""

from __future__ import print_function      # dla zapewnienia
kompatybilności 2.x

seplen = 60
sepchr = '-'

def listing(module, verbose=True):
    sepline = sepchr * seplen
    if verbose:
        print(sepline)
        print('nazwa:', module.__name__, 'plik:', module.__file__)
        print(sepline)
    count = 0
```

```

        for attr in sorted(module.__dict__):           # Przeszukanie (lub
wyliczenie) kluczy przestrzeni nazw
            print('%02d %s' % (count, attr), end = ' ')
            if attr.startswith('__'):
                print('<zmienna wbudowana>')          # Pominięcie __file__ itd.
            else:
                print(getattr(module, attr))           # To samo co __dict__[attr]
            count += 1
        if verbose:
            print(sepline)
            print(module.__name__, 'ma %d zmiennych' % count)
            print(sepline)
    if __name__ == '__main__':
        import mydir
        listing(mydir)                            # Test samosprawdzający –
wyświetlanie samego siebie

```

Warto zwrócić uwagę na notkę dokumentacyjną u góry kodu. Tak jak w poprzednim przykładzie pliku *formats.py*, z uwagi na to, że możemy chcieć użyć modułu jako uniwersalnego narzędzia, dodaliśmy do niego notkę dokumentacyjną, aby informacje na jego temat były dostępne za pośrednictwem funkcji `help` lub przeglądarki systemu PyDoc — narzędzia, które wykorzystuje podobne narzędzia introspekcji do wykonywania swoich zadań. Na końcu kodu tego modułu znajduje się również *autotest* modułu, który w nieco narcystyczny sposób importuje i wyświetla samego siebie. Poniżej zamieszczamy przykładowe wyniki uzyskane w Pythonie 3.3; ten skrypt działa również w wersji 2.x (gdzie może wyświetlać mniej nazw), ponieważ korzystamy z importu `__future__`:

```
c:\code> py -3 mydir.py
-----
name: mydir file: c:\code\mydir.py
-----
00) __builtins__ <built-in name>
01) __cached__ <built-in name>
02) __doc__ <built-in name>
03) __file__ <built-in name>
04) __initializing__ <built-in name>
05) __loader__ <built-in name>
06) __name__ <built-in name>
07) __package__ <built-in name>
08) listing <function listing at 0x000000000295B488>
```

```
09) print_function _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0),
65536)
10) sepchr -
11) seplen 60
-----
mydir has 12 names
-----
```

Aby użyć tego jako narzędzia do wyświetlania listy innych modułów, wystarczy do naszej nowej funkcji przekazać obiekty modułów jako argumenty wywołania. Oto lista atrybutów w module `tkinter` w standardowej bibliotece (np. w Pythonie 2.x moduł ten nosi nazwę Tkinter); technicznie rzecz biorąc, nasz program będzie działał na każdym obiekcie z atrybutami `__name__`, `__file__` i `__dict__`:

```
>>> import mydir
>>> import tkinter
>>> mydir.listing(tkinter)
-----
name: tkinter file: C:\Python33\lib\tkinter\__init__.py
-----
00) ACTIVE active
01) ALL all
02) ANCHOR anchor
03) ARC arc
04) At <function At at 0x0000000002BD41E0>
...many more names omitted...
156) image_types <function image_types at 0x0000000002BE2378>
157) mainloop <function mainloop at 0x0000000002BCBBF8>
158) sys <module 'sys' (built-in)>
159) wantobjects 1
160) warnings <module 'warnings' from 'C:\\Python33\\lib\\warnings.py'>
-----
tkinter has 161 names
-----
```

Z funkcją `getattr` i jej podobnymi spotkamy się ponownie później. Należy tutaj zwrócić uwagę na to, że `mydir` jest programem pozwalającym na przeglądanie innych programów. Ponieważ Python udostępnia mechanizmy wewnętrzne obiektów, możemy je przetwarzać w sposób generyczny[\[2\]](#).

Importowanie modułów z użyciem nazwy w postaci ciągu znaków

Nazwa modułu w instrukcji `import` lub `from` jest zapisaną na stałe nazwą zmiennej. Czasami jednak program otrzyma w czasie wykonywania nazwę importowanego modułu w postaci łańcucha znaków (na przykład kiedy użytkownik wybierze nazwę modułu w GUI lub dokonamy parsowania dokumentu XML). Niestety, nie da się wykorzystać instrukcji `import` do bezpośredniego załadowania modułu, mając podaną jego nazwę w postaci łańcucha znaków. Python oczekuje tutaj nazwy zmiennej, a nie łańcucha znaków czy wyrażenia. Na przykład:

```
>>> import 'string'  
File "<stdin>", line 1  
    import "string"  
          ^  
  
SyntaxError: invalid syntax
```

Nie zadziała również proste przypisanie łańcucha znaków do zmiennej.

```
x = 'string'  
import x
```

Python spróbuje tutaj zaimportować plik `x.py`, a nie moduł `string`. Nazwa podana w instrukcji `import` staje się zmienną przypisaną do załadowanego modułu, a także identyfikuje plik zewnętrzny.

Uruchamianie ciągów znaków zawierających kod

Aby obejść te ograniczenia, musimy skorzystać ze specjalnych narzędzi, które załadują moduł dynamicznie na podstawie nazwy reprezentowanej przez łańcuch znaków generowany w czasie działania programu. Najbardziej uniwersalnym podejściem jest skonstruowanie instrukcji `import` jako łańcucha kodu Pythona i przekazanie go do wbudowanej funkcji `exec` w celu wykonania (`exec` w Pythonie 2.x jest instrukcją, jednak można jej użyć w dokładnie ten sam sposób jak poniżej — nawiasy zostaną po prostu zignorowane):

```
>>> modname = 'string'  
>>> exec('import ' + modname)                      # Wykonanie łańcucha kodu  
>>> string                                         # Zainportowany w tej  
przestrzeni nazw  
  
<module 'string' from 'c:\Python30\lib\string.py'>
```

Funkcję `exec` (i jej odpowiednik dla wyrażeń, funkcję `eval`) poznaliśmy już wcześniej — w rozdziale 3. i rozdziale 10. Kompiluje ona łańcuch znaków zawierający kod i przekazuje go do interpretera Pythona w celu wykonania. W Pythonie kompilator kodu bajtowego jest dostępny w czasie wykonywania, dlatego w taki sposób możemy tworzyć programy generujące i wykonujące inne programy. Domyslnie funkcja `exec` wykonuje kod w bieżącym zasięgu, jednak możemy to zmienić, przekazując opcjonalne słowniki przestrzeni nazw. Ma ona również problemy z bezpieczeństwem, o których pisaliśmy już wcześniej w tej książce, niemniej w przypadku małych fragmentów kodu, które sam tworzysz, są one zwykle mało istotne.

Bezpośrednie wywołania: dwie opcje

Jedyną prawdziwą wadą instrukcji `exec` jest to, że musi ona kompilować instrukcję `import` za każdym wykonaniem, a proces komplikacji może być dosyć wolny. Prekompilowanie do postaci kodu bajtowego za pomocą wbudowanej funkcji `compile` może pomóc w przypadku wielokrotnie wykonywanych fragmentów kodu, ale w większości przypadków będzie prawdopodobnie szybciej i łatwiej użyć wbudowanej funkcji `__import__` do załadowania modułu, którego nazwa jest reprezentowana przez ciąg znaków, o czym pisaliśmy już w rozdziale 22. Efekt będzie podobny, jednak funkcja `__import__` zwraca obiekt modułu, zatem by obiekt ten zachować, trzeba go przypisać do zmiennej.

```
>>> modname = 'string'  
>>> string = __import__(modname)  
>>> string  
<module 'string' from 'c:\\\\Python33\\\\lib\\\\string.py'>
```

Jak wspominaliśmy również w rozdziale 22., nowsze wywołanie `importlib.import_module` wykonuje to samo zadanie i jest ogólnie preferowanym rozwiązaniem w nowszych wersjach Pythona do bezpośredniego importowania na podstawie ciągu znaków reprezentującego nazwę modułu — przynajmniej zgodnie z obecną „oficjalną” polityką określoną w instrukcjach Pythona:

```
>>> import importlib  
>>> modname = 'string'  
>>> string = importlib.import_module(modname)  
>>> string  
<module 'string' from 'C:\\\\Python33\\\\lib\\\\string.py'>
```

Wywołanie funkcji `import_module` pobiera ciąg nazwy modułu i opcjonalny drugi argument, który wskazuje *pakiet* użyty jako punkt zaczepienia dla importu względneego, domyślnie ustawiony na `None`. Wywołanie to działa tak samo jak `__import__` w swoich podstawowych rolach, ale więcej informacji na ten temat znajdziesz w dokumentacji Pythona.

Chociaż oba wywołania nadal działają, w tych wersjach Pythona, w których oba rozwiązania są dostępne, oryginalny `__import__` jest zasadniczo przeznaczony do dostosowywania operacji importowania poprzez zmianę przypisania we wbudowanym zasięgu (a omawianie wszelkich przyszłych zmian w „oficjalnych” zasadach wykracza daleko poza zakres tej książki!).

Przykład — przechodnie przeładowywanie modułów

W tej sekcji opracujemy narzędzie modułowe, co pozwoli połączyć ze sobą kilka wcześniej omawianych tematów i posłuży jako większe studium przypadku do zamknięcia zarówno tego rozdziału, jak i całej części książki. Przeładowywanie modułów omawialiśmy w rozdziale 23. jako sposób pobrania aktualnień kodu bez zatrzymywania i wznawiania działania programu. Kiedy jednak przeładowujemy moduł, Python przeładowuje jedynie plik tego określonego modułu. Nie przeładowuje automatycznie wszystkich modułów, które ten plik importuje.

Na przykład, jeżeli zimportujemy moduł A, a z kolei A importuje moduły B oraz C, operacja przeładowania odnosi się tylko do A, a nie również do B oraz C. Instrukcje wewnętrz modułu A

importujące moduły B i C są w czasie przeładowywania wykonywane raz jeszcze, jednak po prostu pobierają one załadowane obiekty plików B oraz C (zakładając, że zostały one wcześniej zimportowane). W prawdziwym, ale mimo to nieco abstrakcyjnym kodzie plik A.py wyglądałby następująco.

```
#A.py
import B                                # Nie jest przeładowywany razem
z A!
import C                                # Importuje załadowany już
moduł
% python
>>> . . .
>>> from imp import reload
>>> reload(A)
```

Domyślnie oznacza to, że nie należy polegać na tym, iż przeładowywanie pobierze zmiany we wszystkich modułach w sposób przechodni. Zamiast tego należy użyć kilku osobnych wywołań funkcji `reload`, które niezależnie uaktualnią poszczególne komponenty. W przypadku dużych systemów testowanych interaktywnie może to wymagać sporo pracy. Jeżeli jest to pożądane, możemy zaprojektować nasz system w taki sposób, by przeładowywał on swoje komponenty automatycznie, dodając wywołanie `reload` w module nadziednym, takim jak A, jednak komplikuje to kod modułu.

Przeładowywanie rekurencyjne

Lepszym rozwiązaniem jest napisanie uniwersalnego narzędzia, które przeładowania tego typu wykonuje automatycznie, przeglądając atrybuty `__dict__` modułów i sprawdzając typ każdego elementu w celu odnalezienia zagnieżdżonych modułów, które można przeładować. Takie narzędzie mogłoby być funkcją pomocniczą wywołującą samą siebie w sposób *rekurencyjny*, co pozwoliłoby jej przetwarzać dowolnie złożone łańcuchy zależnych od siebie importów. Atrybuty `__dict__` modułów zostały wprowadzone w rozdziale 23. i zastosowane na początku bieżącego rozdziału, natomiast wywołanie `type` zostało zaprezentowane w rozdziale 9.; teraz musimy tylko połączyć ze sobą te dwa narzędzia.

Pokazany poniżej moduł `reloadall.py` zawiera funkcję `reload_all`, która automatycznie przeładowuje dany moduł, każdy moduł importowany przez ten moduł i tak dalej aż do końca każdego łańcucha importowanych plików. Wykorzystuje słownik do przechowywania informacji o przeładowanych już modułach, a także rekurencję, która pozwala przechodzić przez łańcuch modułów. Oprócz tego w kodzie zastosowano moduł `types` z biblioteki standardowej, który z wyprzedzeniem definiuje wyniki wywołania `type` dla typów wbudowanych. Technika ze słownikiem `visited` została tutaj wykorzystana w celu uniknięcia cykli w przypadku, gdy importowanie jest rekurencyjne lub zbędne, ponieważ obiekty modułów są mutowalne, dzięki czemu mogą być kluczami słownika (jak wiemy z rozdziałów 5. i 8., podobną funkcjonalność dałby nam `zbiór`, gdybyśmy do wstawiania elementów użyli `visited.add(module)`).

```
#!python
"""
reloadall.py: przechodnie przeładowanie zagnieżdżonych modułów (2.x +3.x)
Wywołaj reload_all z jednym lub większą liczbą importowanych obiektów modułów
"""

reloadall.py: przechodnie przeładowanie zagnieżdżonych modułów (2.x +3.x)
Wywołaj reload_all z jednym lub większą liczbą importowanych obiektów modułów
"""
```

```

import types
from imp import reload
def status(module):
    print('przeładowanie' + module.__name__)
def tryreload(module):
    try:
        reload(module)
    except:
        print('Niepowodzenie: %s' % module)
def transitive_reload(module, visited):
    if not module in visited:
        status(module)
        tryreload(module)
    visited[module] = True
    for attrobj in module.__dict__.values():
        if type(attrobj) == types.ModuleType:
            transitive_reload(attrobj, visited)
def reload_all(*args):
    visited = {}
    for arg in args:
        if type(arg) == types.ModuleType:
            transitive_reload(arg, visited)
def tester(reloader, modname):
    import importlib, sys
    if len(sys.argv) > 1: modname = sys.argv[1]
    module = importlib.import_module(modname)
    reloader(module)
if __name__ == '__main__':

```

```
    tester(reload_all, 'reloadall')                      # autotest: przeładować swój
moduł?
```

Poza słownikami przestrzeni nazw nasz skrypt używa innych narzędzi, które tutaj omawialiśmy: sprawdza atrybut `__name__` w celu uruchomienia autotestu, gdy moduł jest wywoływany jako skrypt najwyższego poziomu, a jego funkcja testująca używa `sys.argv` do sprawdzania argumentów wywołania z poziomu wiersza poleceń oraz `importlib`, aby zaimportować moduł według ciągu znaków reprezentującego nazwę, przekazanego jako argument funkcji lub wiersza polecenia. Mała ciekawostka: zwróć uwagę, w jaki sposób wywołanie funkcji `reload` zostało „opakowane” w instrukcji `try`, aby przechwytywanie wyjątków stało się możliwe — w Pythonie 3.3 przeładowania czasami kończą się niepowodzeniem z powodu zaimplementowania zmodyfikowanego mechanizmu importującego. Instrukcja `try` została wprowadzona w rozdziale 10., a więcej szczegółowych informacji na jej temat znajdziesz w części VII.

Testowanie przeładowań rekurencyjnych

Aby skorzystać z tego narzędzia, powinieneś zaimportować jego funkcję `reload_all` i przekazać do niej nazwę załadowanego już modułu (tak samo jak robilibyśmy to dla wbudowanej funkcji `reload`). Kiedy plik jest wykonywany samodzielnie, jego kod samosprawdzający wywołuje funkcję `reload_all` automatycznie, domyślnie przeładowując swój własny moduł, jeżeli w wierszu wywołania nie zostały podane żadne inne argumenty. W tym trybie moduł musi zaimportować samego siebie, ponieważ jego własna nazwa nie jest zdefiniowana w pliku bez operacji importowania. Kod ten działa zarówno w wersji 3.x, jak i 2.x, ponieważ w wywołaniu `print` użyliśmy znaków + oraz % zamiast przecinka, choć zestaw użytych i, co za tym idzie, przeładowywanych modułów może się różnić w zależności od wiersza wywołania:

```
C:\code> c:\Python33\python reloadall.py
przeładowanie reloadall
przeładowanie types

C:\code> c:\Python27\python reloadall.py
przeładowanie reloadall
przeładowanie types
```

Jeżeli w wierszu polecenia podamy argument wywołania reprezentujący nazwę wybranego modułu, nasz program zamiast samego siebie przeładowuje moduł o podanej nazwie — w poniższym przykładzie będzie to moduł do sprawdzania wydajności kodu, który utworzyliśmy w rozdziale 21. Zauważ, że w tym trybie podajemy nazwę modułu, a nie nazwę pliku (tak jak w przypadku instrukcji `import`, również tu nie dodajczamy rozszerzenia `.py`); po uruchomieniu skrypt importuje moduł za pośrednictwem ścieżki wyszukiwania modułów, tak jak zawsze:

```
c:\code> reloadall.py pybench
reloading pybench
reloading timeit
reloading itertools
reloading sys
reloading time
reloading gc
reloading os
reloading errno
```

```
reloading ntpath
reloading stat
reloading genericpath
reloading copyreg
```

Prawdopodobnie najczęściej będziemy używać naszego nowego programu z poziomu sesji interaktywnej Pythona. Poniżej znajduje się przykład działania tego kodu w Pythonie 3.3 na wybranych modułach biblioteki standardowej. Warto zwrócić uwagę na to, w jaki sposób moduł `os` jest importowany przez bibliotekę `tkinter`; `tkinter` dociera do modułu `sys`, zanim `os` będzie to w stanie zrobić (jeżeli chciałbyś przetestować ten kod w Pythonie 2.x, powinieneś zamiast `tkinter` użyć biblioteki `Tkinter`).

```
>>> from reloadall import reload_all
>>> import os, tkinter
>>> reload_all(os)                                     # Normalny tryb użycia
przeładowanie os
przeładowanie ntpath
przeładowanie stat
przeładowanie sys
przeładowanie genericpath
przeładowanie errno
przeładowanie copyreg
>>> reload_all(tkinter)
przeładowanie tkinter
przeładowanie _tkinter
przeładowanie warnings
przeładowanie sys
przeładowanie linecache
przeładowanie tokenize
przeładowanie builtins
FAILED: <module 'builtins'>
przeładowanie re
...itd...
przeładowanie os
przeładowanie ntpath
przeładowanie stat
przeładowanie genericpath
przeładowanie errno
```

...itd...

A oto jeszcze jedna sesja zestawiająca efekt działania normalnego i przechodniego przeładowywania. Zmiany wprowadzone w dwóch zagnieżdżonych plikach nie są uwzględniane w przeładowaniach dopóty, dopóki nie skorzystamy z narzędzia do przeładowywania przechodniego:

```
import b                                # Plik a.py
X = 1
import c                                # Plik b.py
Y = 2
Z = 3                                    # Plik c.py
C:\code> py -3
>>> import a
>>> a.x, a.b.Y, a.b.c.Z
(1, 2, 3)

# Bez zatrzymywania Pythona zmień wartości przypisanych we wszystkich plikach i
# zapisz zmiany
>>> from imp import reload
>>> reload(a)                            # Wbudowana funkcja reload działa
tylko na najwyższym poziomie
<module 'a' from '.\\a.py'>
>>> a.x, a.b.Y, a.b.c.Z
(111, 2, 3)
>>> from reloadall import reload_all
>>> reload_all(a)                      # Normalny tryb użytkowania
przeładowanie a
przeładowanie b
przeładowanie c
>>> a.x, a.b.Y, a.b.c.Z                # Przeładowuje również wszystkie
zagnieżdżone moduły
(111, 222, 333)
```

Uważnie przeanalizuj kod tego przykładu i jego wyniki, aby lepiej poznać działanie mechanizmu przeładowywania modułów. W kolejnym podrozdziale będziemy rozbudowywać przedstawione tutaj narzędzia.

Rozwiązania alternatywne

Dla wszystkich fanów rekurencji poniżej przedstawiamy alternatywne rozwiązanie *rekurencyjne* dla funkcji z poprzedniej sekcji, które do wykrywania cykli zamiast słownika używa *zbiorów*, jest nieco bardziej *bezpośrednie*, ponieważ eliminuje pętlę najwyższego poziomu, i znakomicie nadaje się do zilustrowania ogólnego sposobu działania funkcji rekurencyjnych (porównaj je z

oryginałem, aby zobaczyć różnice). Ta nowa wersja wykorzystuje pewne fragmenty kodu z oryginału, chociaż kolejność ponownego ładowania modułów może się różnić, jeżeli zmienia się kolejność elementów słownika przestrzeni nazw:

```
#####
reloadall2.py: przeładowanie zagnieżdżonych modułów (wersja alternatywna)
#####

import types

from imp import reload                                # instrukcja from jest
wymagana w wersji 3.x

from reloadall import status, tryreload, tester

def transitive_reload(objects, visited):
    for obj in objects:
        if type(obj) == types.ModuleType and obj not in visited:
            status(obj)
            tryreload(obj)                            # Przeładowanie bieżącego
modułu i rekurencja
            visited.add(obj)
            transitive_reload(obj.__dict__.values(), visited)

def reload_all(*args):
    transitive_reload(args, set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall2')                # Autotest: przeładować swój
moduł?
```

Jak pokazywaliśmy w rozdziale 19., dla większości funkcji rekurencyjnych istnieją odpowiedniki wykorzystujące *jawny stos* (ang. *explicit stack*) lub kolejki, co może być preferowane w niektórych zastosowaniach. Poniżej przedstawiamy jeden z takich programów przeładowujących moduły, który używa wyrażenia generatora, aby odfiltrować elementy niebędące modułami oraz moduły z przestrzeni nazw bieżącego modułu, które zostały już „odwiedzone”. Ponieważ kod skryptu zarówno pobiera, jak i dodaje elementy na końcu listy, jest oparty na stosie, a kolejność przeładowywania modułów jest determinowana przez kolejność nazw i elementów słownika — skrypt „odwiedza” kolejne moduły w słownikach przestrzeni nazw od prawej do lewej, w przeciwieństwie do kolejności od lewej do prawej w rekurencyjnych wersjach tego narzędzia (spróbuj samodzielnie przeanalizować kod, aby dowiedzieć się, jak działa). Możemy próbować to zmieniać, ale kolejność elementów w słowniku i tak jest dowolna.

```
#####
reloadall3.py: przeładowanie zagnieżdżonych modułów (wersja z
wykorzystaniem jawnego stosu)
#####

import types

from imp import reload                                # instrukcja from jest
wymagana w wersji 3.x
```

```

from reloadall import status, tryreload, tester
def transitive_reload(modules, visited):
    while modules:
        next = modules.pop()                                # Usuń kolejny element z
        końca listy
        status(next)                                       # Przeładuj moduł
        tryreload(next)
        visited.add(next)
        modules.extend(x for x in next.__dict__.values())
        if type(x) == types.ModuleType and x not in visited
def reload_all(*modules):
    transitive_reload(list(modules), set())
if __name__ == '__main__':
    tester(reload_all, 'reloadall3')                      # Autotest: przeładować
    swój moduł?

```

Jeżeli rekurencja i brak rekurencji zastosowane w tym przykładzie są mylące, zapoznaj się z omówieniem funkcji rekurencyjnych w rozdziale 19., gdzie znajdziesz dodatkowe informacje na ten temat.

Testowanie wariantów przeładowania

Aby udowodnić, że różne wersje naszego programu działają tak samo, przetestujmy wszystkie trzy warianty. Dzięki wspólnej funkcji testowania możemy uruchomić wszystkie trzy wersje bezpośrednio z poziomu wiersza poleceń, zarówno bez argumentów, aby przetestować samo ładowanie modułu, jak i z argumentem wywołania w postaci nazwy modułu, który ma zostać ponownie załadowany (zostanie przekazany w `sys.argv`):

```

c:\code> reloadall.py
reloading reloadall
reloading types

c:\code> reloadall2.py
reloading reloadall2
reloading types

c:\code> reloadall3.py
reloading reloadall3
reloading types

```

Chociaż trudno to tutaj zobaczyć, naprawdę testujemy poszczególne alternatywy przeładowywania modułów — każdy z tych testów ma taką samą funkcję autotestu, ale przekazuje jej funkcję `reload_all` z własnego pliku. Oto wyniki przeładowywania modułu biblioteki graficznej `tkinter` z wersji 3.x i wszystkich powiązanych z nią modułów:

```

c:\code> reloadall.py tkinter
reloading tkinter

```

```
reloading _tkinter
reloading tkinter._fix
...itd...
c:\code> reloadall2.py tkinter
reloading tkinter
reloading tkinter.constants
reloading tkinter._fix
...itd...
c:\code> reloadall3.py tkinter
reloading tkinter
reloading sys
reloading tkinter.constants
...itd...
```

Wszystkie trzy wersje działają zarówno w Pythonie 3.x, jak i 2.x — sposób wyświetlania komunikatów jest ujednolicony za pomocą formatowania i unikamy używania narzędzi specyficznych dla wersji (choć czasami musisz używać nazw modułów z wersji 2.x, takich jak na przykład Tkinter; do uruchamiania w systemie Windows używam launchera w wersji 3.3 — więcej informacji na ten temat znajdziesz w dodatku B):

```
c:\code> py -2 reloadall.py
reloading reloadall
reloading types
c:\code> py -2 reloadall2.py Tkinter
reloading Tkinter
reloading _tkinter
reloading FixTk
...itd...
```

Jak zwykle możemy również testować nasze skrypty interaktywnie, importując odpowiednie moduły i wywołując główną funkcję przeładowującą z argumentem w postaci obiektu modułu lub wywołując autotest z funkcją przeładowania i ciągiem znaków reprezentującym nazwę modułu:

```
C:\code> py -3
>>> import reloadall, reloadall2, reloadall3
>>> import tkinter
>>> reloadall.reload_all(tkinter)                                # Normalny sposób
użycia
reloading tkinter
reloading tkinter._fix
reloading os
```

```

...itd...

>>> reloadall.tester(reloadall2.reload_all, 'tkinter')      # Narzędzie
testujące

reloading tkinter
reloading tkinter._fix
reloading os

...itd...

>>> reloadall.tester(reloadall3.reload_all, 'reloadall3') # Naśladowanie
autotestu

reloading reloadall3
reloading types

```

Jeżeli przyjrzysz się wcześniej wynikom przeładowań biblioteki `tkinter`, z pewnością zauważysz, że każdy z trzech wariantów skryptu może wyświetlać wyniki w innej *kolejności*; wszystko zależy od kolejności elementów słownika przestrzeni nazw, a w ostatnim przypadku zależy również od kolejności dodawania elementów do stosu. W rzeczywistości w Pythonie 3.3 kolejność przeładowywania modułów dla danej wersji programu może się różnić przy każdym uruchomieniu. Aby upewnić się, że wszystkie trzy wersje przeładowują te same moduły niezależnie od kolejności, w jakiej to robią, możemy użyć zbiorów (lub sortowania), uruchamiając odpowiednie polecenia z poziomu powłoki za pomocą metody `os.popen`, którą poznaliśmy w rozdziale 13. i której później używaliśmy jeszcze w rozdziale 21.:

```

>>> import os

>>> res1 = os.popen('reloadall.py tkinter').readlines()
>>> res2 = os.popen('reloadall2.py tkinter').readlines()
>>> res3 = os.popen('reloadall3.py tkinter').readlines()
>>> res1[:3]
['reloading tkinter\n', 'reloading sys\n', 'reloading tkinter._fix\n']
>>> res1 == res2, res2 == res3
(False, False)
>>> set(res1) == set(res2), set(res2) == set(res3)
(True, True)

```

Gorąco zachęcamy do uruchamiania przedstawionych skryptów, analizowania ich kodu i samodzielnego eksperymentowania, aby lepiej poznać ich sposoby działania; są to rodzaje importowalnych narzędzi, które możesz dodawać do własnej biblioteki kodu źródłowego. Zwróć uwagę na podobną technikę testowania w klasach, które będziemy omawiać w rozdziale 31., gdzie zastosujemy ją do przekazywanych *obiektów klas* i rozbudujemy o nowe możliwości.

Powinieneś również pamiętać, że wszystkie trzy warianty przeładowują tylko moduły, które zostały załadowane za pomocą instrukcji `import` — ponieważ nazwy skopowane z instrukcji `from` nie powodują zagnieźdzania modułów i odwoływanie się do nich w przestrzeni nazw importera, jego moduł zawierający nie jest ponownie ładowany. Zasadniczo mechanizm przechodniego przeładowywania modułów polega na tym, że dany moduł przeładowuje aktualizacje modułów zależnych w *miejscu*, tak że wszystkie odwołania do tych modułów w dowolnym zasięgu automatycznie uzyskają dostęp do zaktualizowanej wersji. Importy z użyciem polecenia `from` kopią nazwy, zatem nie są one później aktualizowane przez przeładowania

(przechodnie lub nie) — a ich obsługa może wymagać uważnej analizy kodu źródłowego lub dostosowania operacji importowania (więcej wskazówek znajdziesz w rozdziale 22.).

Wpływ przeładowywania na moduły może być kolejnym powodem, dla którego bardziej wolimy używać instrukcji `import` niż `from` — co prowadzi nas do końca tego rozdziału i kolejnej części książki oraz standardowego zestawu ostrzeżeń i pułapek związanych z omawianymi w niej zagadnieniami.

Pułapki związane z modułami

W tej sekcji przyjrzymy się tradycyjnemu zbiorowi przypadków granicznych, które sprawiają, że życie początkujących programistów Pythona nabiera barw. Niektóre są tak dziwne, że wymyślenie dla nich jakiegoś reprezentatywnego przykładu może być prawdziwym wyzwaniem, jednak większość ilustruje takie czy inne, ale niezmiennie ważne cechy języka Python.

Kolizje nazw modułów: pakiety i importowanie względne w pakietach

Jeżeli masz dwa moduły o tej samej nazwie, możesz być w stanie zaimportować tylko jeden z nich — domyślnie zawsze wybierany będzie ten, którego katalog znajduje się najbardziej na lewo w ścieżce wyszukiwania modułów `sys.path`. Nie stanowi to problemu, jeżeli preferowany moduł znajduje się w katalogu skryptu najwyższego poziomu; ponieważ taki katalog jest zawsze pierwszy na ścieżce modułu, jego zawartość zostanie zlokalizowana automatycznie. Jednak w przypadku importu z wielu katalogów liniowy charakter ścieżki wyszukiwania modułów oznacza, że pliki o tej samej nazwie mogą ze sobą kolidować.

Aby to naprawić, unikaj stosowania plików o tych samych nazwach lub skorzystaj z funkcji importowania pakietów, omawianych w rozdziale 24. Jeżeli musisz użyć obu plików o tej samej nazwie, uporządkuj pliki źródłowe w podkatalogach, tak aby nazwy katalogów importu pakietów tworzyły unikatowe odniesienia do modułów. Dopóki nazwy katalogów pakietów pozostaną unikalne, będziesz mieć dostęp do jednego lub obu modułów o tej samej nazwie.

Zauważ, że ten problem może pojawić się również, jeżeli przypadkowo użyjesz nazwy własnego modułu, która jest taka sama jak nazwa modułu ze standardowej biblioteki, którego potrzebujesz — w takiej sytuacji moduł lokalny w katalogu domowym programu (lub innym katalogu znajdująącym się na początku ścieżki modułu) może ukryć i zastąpić moduł biblioteki.

Aby to naprawić, unikaj używania takich samych nazw modułów lub przechowuj swoje moduły w katalogu pakietów i użyj modelu importowania względnego pakietów, znanego z wersji 3.x, ale dostępnego również w wersji 2.x. W tym modelu normalne importowanie pomija katalog pakietów (więc dostaniesz moduł z biblioteki), ale istnieją specjalne instrukcje importowania z kropkami, które nadal pozwalają Ci wybrać lokalną wersję modułu, jeżeli to konieczne.

W kodzie najwyższego poziomu kolejność instrukcji ma znaczenie

Jak mogłeś się już wcześniej przekonać, kiedy moduł jest importowany po raz pierwszy (lub przeładowywany), Python wykonuje jego instrukcje jedna po drugiej, od góry pliku do dołu. Ma to kilka konsekwencji związanych z referencjami, o których należy wspomnieć.

- Kod na *najwyższym poziomie* pliku modułu (niezaginieźdżony w funkcji) wykonywany jest od razu, kiedy tylko Python dotrze do niego w czasie importowania; z tego powodu nie

- może się odnosić do zmiennych przypisanych *niżej* w pliku (w dalszej części pliku).
- Kod wewnętrz ciała *funkcji* nie jest wykonywany, dopóki funkcja nie zostanie wywołana. Ponieważ zmienne z funkcji nie są obliczane, dopóki funkcja nie zostanie faktycznie wykonana, zazwyczaj mogą odnosić się do zmiennych znajdujących się *w dowolnym miejscu* pliku.

Zasadniczo referencje odnoszące się do zmiennych zdefiniowanych niżej (odwołania do przodu) są problemem jedynie w kodzie najwyższego poziomu modułu, który wykonywany jest natychmiast w czasie importowania. Funkcje mogą odwoływać się do dowolnych zmiennych. Poniżej znajduje się przykład ilustrujący to zjawisko.

```

func1()                                # Błąd: "func1" nie została jeszcze
przypisana

def func1():
    print(func2())                      # OK: "func2" zostanie wyszukana później

func1()                                # Błąd: "func2" nie została jeszcze
przypisana

def func2():
    return "Witam"

func1()                                # OK: "func1" i "func2" zostały już
przypisane

```

Kiedy powyższy plik zostanie zimportowany (lub wykonany jako samodzielny program), Python wykonuje jego instrukcje od góry do dołu. Pierwsze wywołanie `func1` nie powiedzie się, ponieważ instrukcja `def` dla `func1` nie była jeszcze wykonana. Wywołanie `func2` wewnętrz `func1` działa, jeśli tylko Python dotrze do instrukcji `def` dla `func2` przed wywołaniem funkcji `func1` (a tak nie było, kiedy wykonywane było drugie wywołanie `func1`). Ostatnie wywołanie `func1` na dole pliku działa, ponieważ zmienne `func1` oraz `func2` zostały już przypisane.

Mieszanie instrukcji `def` z kodem najwyższego poziomu jest nie tylko trudne do odczytania, ale również zależne od kolejności instrukcji. Z reguły jeśli musimy mieszać kod wykonywany natychmiast z instrukcjami `def`, instrukcje `def` umieszcza się na górze pliku, a kod najwyższego poziomu na dole. W ten sposób funkcje na pewno zostaną zdefiniowane i przypisane, zanim wykonany zostanie kod je wykorzystujący.

Instrukcja `from` kopiuje nazwy, jednak łącza już nie

Choć instrukcja `from` jest powszechnie używana, jest także w Pythonie źródłem różnych potencjalnych pułapek. Instrukcja ta jest tak naprawdę przypisaniem do zmiennych w zakresie kodu importującego — operacją kopowania nazw, a nie tworzenia do nich aliasów. Implikacje takiego rozwiązania są takie same jak dla wszystkich operacji przypisania w Pythonie, jednak dość subtelne, w szczególności w przypadku kodu współdzielącego obiekty istniejące w różnych plikach. Założymy na przykład, że definiujemy następujący moduł o nazwie `nested1.py`.

```

# Plik nested1.py
X = 99

def printer(): print(X)

```

Jeśli zimportujemy jego dwie zmienne za pomocą instrukcji `from` użytej w innym module (`nested2.py`), otrzymamy kopie zmiennych, a nie łącza do nich. Modyfikacja zmiennej w module importującym zmienia wiązanie jedynie lokalnej wersji tej zmiennej, a nie zmiennej znajdującej się w module `nested1.py`.

```

# Plik nested2.py

from nested1 import X, printer          # Kopiuje zmienne
X = 88                                 # Modyfikuje tylko X lokalne!
printer()                               # X w nested1 nadal ma wartość
99

% python nested2.py

99

```

Jeżeli instrukcję `import` wykorzystamy do pobrania całego modułu, a następnie przypiszemy coś do zmiennej ze składnią kwalifikującą, modyfikujemy oryginalną zmienną z pliku `nested1.py`. Kwalifikacja atrybutów kieruje Pythona do zmiennej w obiekcie modułu, a nie do zmiennej w pliku importującym (`nested3.py`).

```

# Plik nested3.py

import nested1                         # Pobranie modułu jako całości
nested1.x = 88                          # OK: modyfikuje X z modułu
nested1

nested1.printer()

% python nested3.py

88

```

Instrukcja `from *` może zacienić znaczenie zmiennych

Wspomniałem o tym wcześniej, ale z pominięciem szczegółów. Ponieważ kiedy używamy instrukcji `from *`, nie wymieniamy zmiennych, które chcemy zaimportować, możemy w ten sposób przypadkowo nadpisać zmienne już wykorzystywane w zakresie modułu importującego. Co gorsza, może nam być trudno ustalić, skąd wzięła się jakaś zmienna. Jest tak szczególnie wtedy, gdy forma `from *` jest wykorzystywana na większej liczbie importowanych plików niż jeden.

Jeżeli na przykład użyjemy instrukcji `from *` na trzech modułach (tak jak to pokazano poniżej), nie będziemy wiedzieli, co tak naprawdę znaczy wywołanie danej funkcji, o ile nie przeszukamy wcześniej wszystkich trzech modułów zewnętrznych (z których każdy może się znajdować w innym katalogu).

```

>>> from module1 import *                  # Źle: może po cichu nadpisać
nasze zmienne

>>> from module2 import *                  # Gorzej: skąd mamy wiedzieć, co
dostajemy?

>>> from module3 import *

>>> . . .

>>> func()                                # Że co???

```

Rozwiązanie tego problemu jest proste — nie należy tak robić. W instrukcjach `from` trzeba spróbować w jawnym sposobie wymieniać listę atrybutów, które chcemy zaimportować, a także ograniczyć użycie instrukcji `from *` do jednego importowanego modułu na plik. W ten sposób wszystkie niezdefiniowane zmienne muszą należeć do modułu podanego w tej jednej instrukcji `from *`. Możemy zresztą całkowicie pozbyć się tego problemu, jeśli zamiast `from` zawsze

będziemy używali instrukcji `import`, jednak taka rada jest zbyt surowa. Tak jak wszystkie inne elementy języka programowania, `from` jest przydatnym narzędziem, jeśli używa się go z rozważą. Nawet powyższy przykład nie jest zresztą całkowicie niedopuszczalny — program może wykorzystywać tę technikę w celu zebrania zmiennych w jednym miejscu dla wygody, o ile wszyscy o tym wiedzą.

Funkcja `reload` może nie mieć wpływu na obiekty importowane za pomocą `from`

A oto kolejna pułapka związana z wykorzystywaniem instrukcji `from`. Jak wspominaliśmy wcześniej, ponieważ instrukcja `from` po wykonaniu kopiuje (przypisuje) zmienne, nie istnieje żadne łącze z powrotem do modułu, z którego pochodzi dana zmienna. Zmienne zimportowane za pomocą `from` stają się po prostu referencjami do obiektów, do których odnoszą się również zmienne o tych samych nazwach w pliku importowanym.

Ze względu na to zachowanie przeładowanie importowanego modułu nie ma żadnego wpływu na moduły, które zimportowały jego zmienne za pomocą instrukcji `from`. Oznacza to, że zmienne klienta nadal będą się odnosiły do oryginalnych obiektów pobranych za pomocą `from`, nawet jeśli zmienne te w oryginalnym module mają teraz inną wartość.

```
from module import X                                # X może nie odzwierciedlać
przeładowanych modułów!

. .

from imp import reload
reload(module)                                     # Modyfikacja modułu, ale nie
zmiennych

X                                                 # Nadal odnosi się do starego
objektu
```

Aby przeładowywanie było bardziej efektywne, należy zamiast `from` użyć instrukcji `import` oraz składni kwalifikującej zmienną. Ponieważ kwalifikacja zawsze oznacza przejście do modułu, w ten sposób po przeładowaniu odnalezione zostaną nowe wiązania w zmiennych modułu.

```
import module                                    # Pobranie modułu, nie zmiennych

. .

from imp import reload
reload(module)                                     # Modyfikuje moduł w miejscu
module.x                                         # Bieżące X: odzwierciedla
przeładowanie modułu
```

W związku z tym nasze programy przeładowujące moduły, opisane wcześniej w tym rozdziale, nie będą miały żadnego wpływu na nazwy pobrane za pomocą instrukcji `from`, a jedynie na te pobrane za pomocą `import`. Wynika stąd zatem, że jeżeli w swoim programie zamierzasz korzystać z przeładowań, prawdopodobnie lepszym rozwiązaniem będzie pozostanie przy używaniu instrukcji `import`.

Funkcja `reload` i instrukcja `from` a testowanie interaktywne

Tak naprawdę problem zasygnalizowany powyżej może być jeszcze bardziej subtelny, niż mogłoby się wydawać. W rozdziale 3. ostrzegałem, że zazwyczaj lepiej jest nie uruchamiać programów za pomocą importowania i przeładowywania ze względu na związane z takim rozwiązaniem trudności. Wszystko jeszcze się pogarsza, kiedy do tej techniki dołożymy instrukcję `from`. Osoby początkujące często napotykają takie problemy w sytuacjach jak opisana poniżej. Założymy, że po otwarciu pliku modułu w oknie edytora tekstu uruchamiamy sesję interaktywną, która ładuje moduł za pomocą `from` w celu przetestowania go.

```
from module import function  
function(1, 2, 3)
```

Po odnalezieniu błędu wracamy do okna edytora, wprowadzamy poprawkę i próbujemy w następujący sposób przeładować moduł.

```
from imp import reload  
reload(module)
```

Takie rozwiązanie jednak nie działa, ponieważ instrukcja `from` przypisała zmienną `function`, a nie `module`. By odwołać się do modułu z funkcji `reload`, musimy przynajmniej raz załadować ten moduł za pomocą instrukcji `import`.

```
from imp import reload  
import module  
reload(module)  
function(1, 2, 3)
```

Takie rozwiązanie również nie działa — funkcja `reload` uaktualnia obiekt modułu, ale, zgodnie z informacjami przedstawionymi wyżej, zmienne takie jak `function`, skopiowane z modułu w przeszłości, nadal odnoszą się do *starych wersji obiektów* (w tym przypadku oryginalnej wersji funkcji). By naprawdę otrzymać nową funkcję, musimy odwołać się do niej jako `module.function` po funkcji `reload`, ewentualnie raz jeszcze wykonać instrukcję `from`.

```
from imp import reload  
import module  
reload(module)  
from module import function # Można też się poddać i użyć  
module.function()  
function(1, 2, 3)
```

Dopiero teraz nowa wersja funkcji zostanie wykonana, ale wygląda na to, że kosztowało nas to całkiem sporo pracy.

Jak widać, wykorzystywanie funkcji `reload` w połączeniu z `from` wiąże się z pewnymi problemami — nie tylko musimy pamiętać o przeładowywaniu modułu po jego zaimportowaniu, ale także musimy po przeładowaniu wykonać instrukcję `from` raz jeszcze. Jest to na tyle skomplikowane, że niejeden ekspert ma z tym od czasu do czasu problemy. Tak naprawdę w Pythonie 3.x sytuacja jeszcze się pogorszyła, ponieważ trzeba dodatkowo pamiętać o zaimportowaniu samego `reload`!

W skrócie, nie powinniśmy oczekwać, że funkcja `reload` i instrukcja `from` będą ze sobą dobrze współpracować. Najlepiej jest w ogóle ich ze sobą nie łączyć, `reload` wykorzystywać w połączeniu z `import` lub uruchamiać programy w inny sposób, zgodnie z sugestiami z rozdziału 3. (na przykład korzystając z opcji *Run Module* z menu *Run* środowiska IDLE, klikając ikony programu lub używając systemowego wiersza poleceń czy wbudowanej funkcji `exec`).

Rekurencyjne importowanie za pomocą `from` może nie działać

Najdziwniejszą (i na szczęście rzadko spotykaną) pułapkę zostawiłem na koniec. Ponieważ operacja importowania wykonuje instrukcję pliku od góry do dołu, musimy uważać, kiedy wykorzystujemy moduły, które się wzajemnie importują. Jest to często nazywane importem *rekurencyjnym*, ale rekurencja tak naprawdę tutaj nie występuje (być może lepszym określeniem byłoby nawet importowanie *okrężne*, albo coś w tym stylu) i takie operacje importowania nigdy nie utkną w nieskończonej pętli. Nie zmienia to jednak faktu, że ponieważ nie wszystkie instrukcje z modułu mogą zostać wykonane, kiedy zimportuje on inny moduł, niektóre zmienne mogą jeszcze nie istnieć.

Jeżeli wykorzystujemy instrukcję `import` do pobrania modułu w całości, najprawdopodobniej nie będzie to miało żadnego znaczenia. Dostęp do zmiennych będzie miał miejsce dopiero przy późniejszym zastosowaniu składni kwalifikującej do pobrania ich wartości, a do tego czasu kod modułu zazwyczaj będzie już wykonany w całości. Jeżeli jednak w celu pobrania określonych zmiennych użyjesz `from`, musisz pamiętać, że będziesz miał dostęp jedynie do tych zmiennych modułu, które zostały już przypisane.

Rozważmy na przykład dwa poniższe moduły o nazwach `recur1` i `recur2`. Moduł `recur1` przypisuje zmienną `X`, a następnie importuje moduł `recur2` przed przypisaniem zmiennej `Y`. W tym momencie `recur2` może pobrać `recur1` jako całość za pomocą instrukcji `import` (moduł ten istnieje już wewnętrznej tabeli modułów Pythona), jeśli jednak używa `from`, zobaczy tylko zmienną `X`. Zmienna `Y` przypisana poniżej instrukcji `import` w `recur1` jeszcze nie istnieje, dlatego otrzymamy błąd.

```
# Plik recur1.py
X = 1
import recur2
# Wykonuje recur2, jeśli obiekt
# modułu nie istnieje
Y = 2
# Plik recur2.py
from recur1 import X
# OK: "X" jest już przypisane
from recur1 import Y
# Błąd: "Y" nie jest przypisane
C:\code> py -3
>>> import recur1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File ".\recur1.py", line 2, in <module>
      import recur2
    File ".\recur2.py", line 2, in <module>
      from recur1 import Y
ImportError: cannot import name Y
```

Python unika ponownego wykonywania instrukcji modułu `recur1`, kiedy jest on importowany rekurencyjnie z `recur2` (w przeciwnym razie operacja importowania powodowałaby nieskończoną pętlę skryptu, której przerwanie wymagałoby użycia kombinacji klawiszy *Ctrl+C*)

lub czegoś jeszcze gorszego). W momencie zimportowania przez `recur2` przestrzeń nazw modułu `recur1` jest jednak niekompletna.

Rozwiążanie? Nie używaj instrukcji `from` w operacjach importowania rekurencyjnego (naprawdę!). Jeżeli jednak tak zrobisz, Python nie zablokuje się co prawda w cyklu, ale Twoje programy znowu będą uzależnione od kolejności instrukcji w modułach. W praktyce istnieją dwa wyjścia z takiej sytuacji:

- Zazwyczaj można wyeliminować takie cykle importowania za pomocą rozważnego projektowania — maksymalizacja spójności i minimalizacja połączeń to dobre pierwsze kroki.
- Jeżeli nie możemy całkowicie przerwać cykli, należy opóźnić dostęp do nazwy modułu za pomocą wykorzystania instrukcji `import` oraz składni kwalifikującej (zamiast stosowania `from`) lub wykonania instrukcji `from` albo wewnątrz funkcji (zamiast na najwyższym poziomie modułu), albo na dole pliku, tak by opóźnić ich wykonanie.

Istnieje jeszcze jedno rozwiązanie, które poznasz w jednym z ćwiczeń na końcu tego rozdziału — i do tego miejsca właśnie oficjalnie dotarliśmy.

Podsumowanie rozdziału

W niniejszym rozdziale omówiono niektóre bardziej zaawansowane koncepcje związane z modułami. Zapoznaliśmy się między innymi z technikami ukrywania danych, z włączaniem nowych możliwości języka za pomocą modułu `__future__`, ze zmienną trybu użycia `__name__` czy przeładowywaniem przechodnim i importowaniem po łańcuchu znaków nazwy. Przyjrzaliśmy się również różnym zagadnieniom związanym z projektowaniem modułów, a także często popełnianym błędem związanym z modułami, co powinno pomóc nam uniknąć popełniania ich we własnym kodzie.

Kolejny rozdział rozpoczyna nasze omówienie narzędzia programowania zorientowanego obiektywo w Pythonie — klasy. Większość informacji z ostatnich kilku rozdziałów będzie miała zastosowanie również tutaj — klasy istnieją w modułach i są przestrzeniami nazw, jednak dodają do wyszukiwania atrybutów dodatkowy komponent o nazwie „wyszukiwanie dziedziczenia”. Ponieważ jest to ostatni rozdział tej części książki, zanim zagłębimy się w ten nowy temat, warto zapoznać się ze zbiorem ćwiczeń końcowych. Najpierw jednak pora na quiz podsumowujący informacje przedstawione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Co specjalnego jest w zmiennych znajdujących się na najwyższym poziomie modułu, rozpoczynających się od pojedynczego znaku podkreślenia?
2. Co oznacza ustalenie zmiennej `__name__` modułu na łańcuch znaków `"__main__"`?
3. Jeżeli użytkownik wpisze w sesji interaktywnej nazwę modułu do przetestowania, w jaki sposób możemy ten moduł zaimportować?
4. Czym różni się modyfikacja `sys.path` od ustawienia `PYTHONPATH` w celu zmodyfikowania ścieżki wyszukiwania modułów?
5. Skoro moduł `__future__` pozwala importować przyszłe opcje języka, czy istnieje jakiś sposób na importowanie opcji z przeszłości?

Sprawdź swoją wiedzę – odpowiedzi

1. Zmienne na najwyższym poziomie pliku modułu, których nazwy rozpoczynają się od pojedynczego znaku `_`, nie są kopiowane do zakresu importującego, kiedy używamy instrukcji `from *`. Można jednak uzyskać do nich dostęp za pomocą instrukcji `import` czy normalnej postaci instrukcji `from`. Lista `__all__` jest podobna, ale logicznie odwrotna; jej zawartość to jedyne nazwy kopowane z `*`.
2. Kiedy zmienna `__name__` modułu jest łańcuchem znaków "`__main__`", oznacza to, że plik jest wykonywany jako skrypt najwyższego poziomu, a nie importowany z innego pliku programu. Inaczej mówiąc, plik ten jest wykorzystywany jako program, a nie biblioteka. Taki mechanizm pozwala na wykorzystywanie kodu jako modułu i jako programu jednocześnie, a także umożliwia zaimplementowanie autotestów.
3. Dane od użytkownika zazwyczaj trafiają do skryptu w postaci łańcucha znaków. By zaimportować podany moduł, kiedy mamy łańcuch znaków jego nazwy, można utworzyć i wykonać instrukcję `import` za pomocą `exec` lub przekazać łańcuch znaków nazwy do wywołania funkcji `__import__` lub `importlib.import_module`.
4. Modyfikacja `sys.path` ma wpływ jedynie na działający program i jest tymczasowa — zmiana znika, kiedy program kończy swe działanie. Ustawienia `PYTHONPATH` zachowywane są w systemie operacyjnym i są globalnie odczytywane przez wszystkie programy znajdujące się na komputerze. Modyfikacja tych ustawień jest trwała i wykracza poza czas działania programu.
5. Nie, nie da się importować opcji Pythona dostępnych w przeszłości. Możemy instalować (lub uparcie wykorzystywać) starsze wersje języka, jednak najnowszy Python jest zazwyczaj najlepszym Pythonem (przynajmniej w danej edycji, o czym świadczy nieco zaskakująca długowieczność linii 2.x!).

Sprawdź swoją wiedzę – ćwiczenia do części piątej

Rozwiązania ćwiczeń znajdują się w podrozdziale „Część V. Moduły i pakiety” w dodatku D.

1. *Podstawy importowania.* Napisz program zliczający wiersze oraz znaki pliku (podobnie do polecenia `wc` w systemie Unix). Za pomocą edytora tekstu utwórz moduł Pythona o nazwie `mymod.py` eksportujący trzy zmienne najwyższego poziomu:
 - Funkcję `countLines(name)` odczytującą plik wejściowy i zliczającą liczbę znajdujących się w nim wierszy (wskaźówka: większość pracy wykonana za nas metoda `file.readlines`; resztą zajmie się `len`, choć możesz liczyć również na pętle `for` oraz iteratory plików, pozwalające na przetwarzanie plików o ogromnych rozmiarach).
 - Funkcję `countChars(name)` wczytującą plik wejściowy i zliczającą liczbę znajdujących się w nim znaków (wskaźówka: metoda `file.read` zwraca jeden łańcuch znaków, który może być używany w podobny sposób).
 - Funkcję `test(name)` wywołującą obie funkcje zliczające dla określonego pliku wejściowego. Nazwa pliku może zostać przekazana, wpisana na stałe, podana za pomocą wbudowanej funkcji `input` lub pobrana z wiersza poleceń za pomocą listy `sys.argv` zaprezentowanej w przykładzie z `formats.py` i

`reloadall.py` w tym rozdziale. Na początek możemy założyć, że będzie przekazanym argumentem funkcji.

Wszystkie trzy funkcje modułu `mymod` powinny oczekwać podania łańcucha znaków z nazwą pliku. Jeżeli któraś z funkcji będzie miała więcej niż dwa czy trzy wiersze, oznacza to, że próbujesz za dużo zrobić ręcznie — należy skorzystać z podanych wskazówek!

Następnie należy przetestować moduł w sesji interaktywnej, wykorzystując instrukcję `import` oraz referencje atrybutów w celu pobrania wyeksportowanych funkcji. Czy zmienna środowiskowa `PYTHONPATH` musi zawierać katalog, w którym utworzyliśmy moduł `mymod.py`? Należy spróbować wykonać moduł na samym sobie, na przykład za pomocą `test("mymod.py")`. Warto zauważyc, że funkcja `test` otwiera moduł dwukrotnie. Osoby szczególnie ambitne mogą spróbować to naprawić, przekazując obiekt otwartego pliku do dwóch funkcji zliczających (wskazówka: `file.seek(0)` przewija plik).

2. *Instrukcje from i from **. Przetestuj moduł `mymod` z ćwiczenia 1. w sesji interaktywnej, wykorzystując instrukcję `from` do bezpośredniego załadowania eksportowanych funkcji — najpierw po ich nazwach, a później za pomocą wariantu `from *`, który pobiera wszystko.
3. *Wartość __main__*. Do modułu `mymod` dodaj wiersz wywołujący automatycznie funkcję `test` tylko wtedy, gdy moduł uruchamiany jest jako skrypt, a nie importowany. Dodany wiersz będzie najprawdopodobniej sprawdzał wartość `__name__` pod kątem łańcucha znaków "`__main__`", tak jak robiliśmy to w tym rozdziale. Spróbuj wykonać moduł w systemowym wierszu poleceń, a później zaimportować go i przetestować jego funkcje w sesji interaktywnej. Czy nadal działa on w obu trybach?
4. *Zagnieżdżone importowanie*. Utwórz drugi moduł — `myclient.py` — importujący `mymod` i testujący jego funkcje. Następnie wykonaj moduł `myclient` z systemowego wiersza poleceń. Jeżeli `myclient` używa instrukcji `from` do pobrania atrybutów modułu `mymod`, czy funkcje `mymod` będą dostępne z najwyższego poziomu `myclient`? Co będzie, jeśli importujemy moduł za pomocą instrukcji `import`? Spróbuj zapisać w module `myclient` oba warianty i przetestować je interaktywnie, importując ten moduł i badając jego atrybut `__dict__`.
5. *Importowanie pakietów*. Zainportuj plik z pakietu. Po utworzeniu podkatalogu o nazwie `mypkg` zagnieżdzonego w katalogu znajdującym się w ścieżce wyszukiwania importowanych modułów przenieś tam plik `mymod.py` utworzony w trzech pierwszych ćwiczeniach i spróbuj zainportować go za pomocą operacji importowania pakietu `import mypkg.mymod`, a następnie wywołać jego funkcje. Spróbuj również pobrać z niego funkcje zliczające za pomocą instrukcji `from`.

Żeby takie rozwiązanie zadziałało, będziesz musiał do katalogu, do którego przeniosłeś plik modułu, dodać plik `__init__.py`. Technika ta powinna jednak działać na wszystkich najważniejszych platformach (między innymi z tego powodu Python wykorzystuje znak kropki jako separator katalogów). Utworzony katalog pakietów może po prostu być podkatalogiem tego, w którym aktualnie pracujemy, dzięki czemu zostanie on odnaleziony przez komponent katalogu głównego ścieżki wyszukiwania i nie będziesz musiał tej ścieżki dodatkowo konfigurować. Do pliku `__init__.py` powinieneś dodać jakiś kod i sprawdzić, czy jest on wykonywany przy każdej operacji importu.
6. *Przeładowywania*. Spróbuj poeksperymentować z przeładowywaniem modułów — wykonaj testy z przykładu `changer.py` z rozdziału 23., modyfikując komunikat wywoływanej funkcji, a także jej zachowanie — bez zatrzymywania interpretera Pythona. W zależności od systemu możesz modyfikować moduł `changer` w innym

oknie lub zawiesić interpreter Pythona i edytować plik w tym samym oknie (w systemie Unix skrót *Ctrl+Z* zazwyczaj zawiesza bieżący proces, a polecenie *fg* go wznowia).

7. *Importowanie wzajemne.* W podrozdziale poświęconym pułapkom związanym z importowaniem rekurencyjnym widać było, że importowanie *recur1* powodowało wystąpienie błędu. Jeżeli jednak ponownie uruchomimy Pythona i zaimportujemy *recur2* interaktywnie, błąd ten nie pojawi się, co można samodzielnie sprawdzić. Dlaczego importowanie *recur2* działa, ale *recur1* już nie? Wskazówka: Python przechowuje nowe moduły we wbudowanej tabeli *sys.modules* (słowniku) przed wykonaniem ich kodu. Późniejsze operacje importowania pobierają najpierw moduł z tabeli, bez względu na to, czy jest on „kompletny”, czy też nie. Teraz można spróbować wykonać moduł *recur1* jako plik skryptu najwyższego poziomu za pomocą polecenia *python recur1.py*. Czy otrzymujemy ten sam błąd, który pojawia się przy interaktywnym importowaniu modułu? Dlaczego? Wskazówka: kiedy moduły są wykonywane jako programy, nie są importowane, dlatego ten przypadek daje taki sam efekt jak interaktywne importowanie *recur2*; *recur2* jest pierwszym importowanym modułem. Co się dzieje, kiedy wykonamy *recur2* jako skrypt? Importowanie wzajemne jest spotykane dosyć rzadko i raczej nie jest tak dziwaczne w praktyce. Z drugiej strony, jeżeli rozumiesz, dlaczego może być potencjalnym problemem, to wiesz już bardzo dużo o semantyce importu Pythona.

[1] Jak pamiętasz z sekcji „Inne metody dostępu do zmiennych globalnych” w rozdziale 17., ponieważ funkcja może uzyskać dostęp do zawierającego ją modułu za pomocą tabeli *sys.modules*, można jej również użyć do emulowania efektu instrukcji *global*. Na przykład wynik zastosowania wyrażenia *global X; X = 0* można symulować (choć za pomocą większej ilości kodu!), wpisując wewnątrz funkcji kod *import sys; glob = sys.modules['__name__']; glob.x = 0*. Pamiętaj, że każdy moduł otrzymuje atrybut *__name__* „za darmo”. Jest on widoczny jako globalna nazwa wewnątrz funkcji w module. Opisana sztuczka zapewnia kolejny sposób modyfikacji zarówno zmiennych lokalnych, jak i globalnych o tej samej nazwie wewnątrz funkcji.

[2] Narzędzia takie jak *mydir.listing* mogą być ładowane z wyprzedzeniem do interaktywnej przestrzeni nazw za pomocą importowania ich w pliku, do którego odnosi się zmienna środowiskowa *PYTHONSTARTUP*. Ponieważ kod z pliku startowego działa w interaktywnej przestrzeni nazw (*module __main__*), importowanie wspólnych narzędzi w pliku startowym może nam zaoszczędzić nieco wpisywania. Więcej informacji na ten temat znajduje się w dodatku A.

Część VI Klasy i programowanie zorientowane obiektowo

Rozdział 26. Programowanie zorientowane obiektowo — wprowadzenie

Dotychczas w książce używaliśmy pojęcia „obiekt” bardzo ogólnie. Tak naprawdę kod napisany przez nas dotychczas był *oparty na obiektach* — przekazywaliśmy obiekty w skryptach, wykorzystywaliśmy je w wyrażeniach, wywoływałyśmy ich metody. Aby nasz kod mógł się jednak zaliczać do prawdziwie *zorientowanego obiektowo* (ang. *object-oriented* — OO), nasze obiekty będą musiały uczestniczyć w czymś, co znane jest pod nazwą *hierarchii dziedziczenia*.

Niniejszy rozdział rozpoczyna nasze omówienie *klas* Pythona — struktur kodu i narzędzi wykorzystywanych do implementowania w Pythonie nowych typów obiektów obsługujących dziedziczenie. Klasy są w Pythonie podstawowym narzędziem programowania zorientowanego obiektowo (ang. *object-oriented programming* — OOP), dlatego w tej części książki przyjrzymy się również podstawom tej koncepcji. Programowanie zorientowane obiektowo oferuje inne i często bardziej wydajne sposoby rozumienia programowania, w których kod jest poddawany faktoryzacji w celu zminimalizowania jego powtarzalności. Nowe programy pisze się natomiast, dostosowując istniejący kod, zamiast modyfikować go w miejscu.

W Pythonie klasy tworzone są za pomocą nowej instrukcji `class`. Jak zobaczymy, obiekty zdefiniowane za pomocą klas mogą w dużej mierze wyglądać tak, jak typy wbudowane omówione wcześniej. Tak naprawdę klasy stosują jedynie — i rozszerzają — koncepcje, o których już wspominaliśmy. Są one w przybliżeniu pakietami funkcji, które wykorzystują oraz przetwarzają wbudowane typy obiektów. Klasy są jednak zaprojektowane w taki sposób, by tworzyły nowe obiekty i nimi zarządały. Obsługują również *dziedziczenie* — mechanizm dostosowywania kodu oraz jego ponownego wykorzystywania, który wykracza poza wszystko, co dotychczas widzieliśmy.

Jedna ważna uwaga: w Pythonie programowanie zorientowane obiektowo jest całkowicie opcjonalne i nie musimy z klas korzystać od razu. Tak naprawdę mnóstwo zadań możemy wykonać za pomocą prostszych konstrukcji, takich jak funkcje, a nawet prosty kod skryptu najwyższego poziomu. Ponieważ wykorzystywanie klas wymaga pewnego planowania z wyprzedzeniem, zazwyczaj jest bardziej interesujące dla osób, które pracują w trybie *strategicznym* (zaangażowanych w długofalowe tworzenie produktów) niż tych pracujących w trybie *taktycznym* (gdzie nigdy nie ma za dużo czasu).

Mimo to — jak zobaczymy w tej części książki — klasy okazują się jednym z najbardziej użytecznych narzędzi udostępnianych przez Pythona. Kiedy się ich właściwie używa, mogą znacznie skrócić czas tworzenia oprogramowania. Są one wykorzystywane w popularnych narzędziach Pythona, takich jak API graficznego interfejsu użytkownika tkinter, dlatego większość programistów Pythona uważa, że znajomość klas na podstawowym poziomie może być naprawdę przydatna.

Po co używa się klas

W rozdziałach 4. i 10. wspominałem, że programy „robią coś z różnymi rzeczami”. W uproszczeniu klasy to tylko sposób definiowania nowych rodzajów otych „rzeczy”, odzwierciedlających prawdziwe obiekty w domenie programu. Założymy na przykład, że decydujemy się zaimplementować hipotetycznego robota wytwarzającego pizzę, który posłużył nam za przykład w rozdziale 16. Jeżeli zaimplementujemy go za pomocą klas, można będzie modelować większą część jego prawdziwej struktury oraz relacji. Przydadzą nam się tutaj dwa aspekty programowania zorientowanego obiektowo.

Dziedziczenie

Roboty wytwarzające pizzę są rodzajami robotów, dlatego posiadają cechy właściwe wszystkim robotom. W terminologii programowania zorientowanego obiektowo mówimy, że „dziedziczą” one właściwości po ogólnej kategorii wszystkich robotów. Te wspólne właściwości wystarczy zaimplementować tylko raz dla przypadku ogólnego, a później użyć ponownie częściowo lub w całości dla wszystkich typów robotów tworzonych w przyszłości.

Kompozycja

Roboty wytwarzające pizzę są tak naprawdę zbiorami komponentów, które działają razem jako zespół. Żeby nasz robot odniósł sukces, musi na przykład mieć ramiona zagniatujące ciasto, napęd umożliwiający przetransportowanie pizzy do pieca i tym podobne. W terminologii programowania zorientowanego obiektowo nasz robot jest przykładem kompozycji — zawiera inne obiekty, które aktywuje, by móc wykonywać swoje funkcje. Każdy komponent może być utworzony jako klasa definiująca własne zachowanie i relacje.

Ogólne pojęcia z dziedziny programowania zorientowanego obiektowo, takie jak dziedziczenie oraz kompozycja, mają zastosowanie do każdej aplikacji, którą można rozbić na zbiór obiektów. W typowych systemach GUI interfejsy są na przykład napisane jako zbiór widgetów — przycisków, podpisów i tym podobnych — które są rysowane, kiedy narysowany zostanie ich pojemnik (*kompozycja*). Co więcej, możemy również pisać własne widgety — przyciski z unikalnymi czcionkami, podpisy z nowymi schematami kolorów, które będą wyspecjalizowanymi wersjami bardziej ogólnych narzędzi interfejsu (*dziedziczenie*).

Z punktu widzenia programowania klasy są jednostkami programów Pythona, podobnie jak funkcje oraz moduły. Są kolejnym pojemnikiem służącym do pakowania logiki oraz danych. Tak naprawdę klasy definiują również nowe przestrzenie nazw, podobnie jak moduły. W porównaniu z innymi jednostkami programów, jakie już widzieliśmy, klasy mają trzy kluczowe i wyróżniające cechy, które sprawiają, że są one bardziej użyteczne przy tworzeniu nowych obiektów.

Wiele instancji

Klasy są tak naprawdę fabrykami generującymi jeden lub większą liczbę obiektów. Za każdym razem, gdy wywołujemy klasę, generujemy nowy obiekt z osobną przestrzenią nazw. Każdy obiekt wygenerowany przez klasę ma dostęp do jej atrybutów *oraz* otrzymuje własną przestrzeń nazw dla swoich danych, które różnią się między obiektami. Jest to nieco podobne do zachowywania stanu funkcji między poszczególnymi wywołaniami, o czym pisaliśmy w rozdziale 17., z tym że w klasach jest to bardzo wyraźne i naturalne, a to tylko jedna z zalet klas. Klasy oferują kompletne rozwiązywanie programistyczne.

Dostosowywanie do własnych potrzeb dzięki dziedziczeniu

Klasy obsługują również koncepcję dziedziczenia z programowania zorientowanego obiektowo; klasę możemy rozszerzyć, redefiniując jej atrybuty poza samą klasą w nowych komponentach oprogramowania zakodowanych jako podklasy. Mówiąc bardziej ogólnie, klasy mogą tworzyć hierarchie przestrzeni nazw definiujące zmienne, które będą wykorzystywane przez obiekty utworzone przez klasy tej hierarchii. Takie rozwiązanie obsługuje wiele zachowań obiektów bardziej bezpośrednio niż inne narzędzia.

Przeciążanie operatorów

Udostępniając specjalne metody protokołów, klasy mogą definiować obiekty odpowiadające na rodzaje operacji, jakie widzieliśmy w przypadku typów wbudowanych. Z obiektów utworzonych za pomocą klas można na przykład tworzyć wycinki, można je poddawać konkatenacji czy indeksowaniu. Python udostępnia punkty zaczepienia, które klasy wykorzystują w celu przejęcia i zaimplementowania dowolnej operacji na typie wbudowanym.

U podstaw mechanizmu programowania zorientowanego obiektowo w Pythonie leżą *dwa nieco magiczne elementy*: specjalny pierwszy argument wywołania funkcji (pozwalający na otrzymanie tematu wywołania) i wyszukiwanie dziedziczonych atrybutów (pozwalające na programowanie przez dostosowywanie). Poza tym cała reszta modelu to w dużej mierze tylko funkcje przetwarzające wbudowane typy danych. Chociaż koncepcja programowania zorientowanego obiektowo nie jest zupełnie nowa, dodaje warstwę strukturalną, która pozwala na znacznie lepsze programowanie niż płaskie modele proceduralne. Wraz z narzędziami funkcyjnymi, które poznaliśmy wcześniej, stanowi to duży, abstrakcyjny krok naprzód, znakomicie ułatwiający tworzenie bardziej wyrafinowanych programów.

Programowanie zorientowane obiektowo z dystansu

Zanim zobaczymy, jak to wszystko wygląda na poziomie kodu, chciałbym powiedzieć kilka słów o ogólnych koncepcjach leżących u podstaw programowania zorientowanego obiektowo. Jeżeli dotychczas nie robiliś nic zorientowanego obiektowo, część terminologii wykorzystywana w niniejszym rozdziale może przy pierwszym podejściu brzmieć niepokojąco. Co więcej, uzasadnienie dla tych pojęć może wydawać się dosyć pokrętne, dopóki nie mamy okazji zapoznać się z tym, jak programiści stosują te koncepcje w większych systemach. Programowanie zorientowane obiektowo jest nie tylko technologią, ale i pewnym doświadczeniem.

Wyszukiwanie atrybutów dziedziczonych

Dobra wiadomość jest taka, że programowanie zorientowane obiektowo jest o wiele łatwiejsze do zrozumienia i użycia w Pythonie niż w innych językach, takich jak C++ czy Java. Jako język skryptowy z typami dynamicznymi Python usuwa wiele z bałaganu składniowego oraz stopnia skomplikowania obecnych w programowaniu zorientowanym obiektowo w innych narzędziach. Tak naprawdę większość koncepcji programowania zorientowanego obiektowo w Pythonie sprowadza się do jednego wyrażenia.

`obiekt.atrybut`

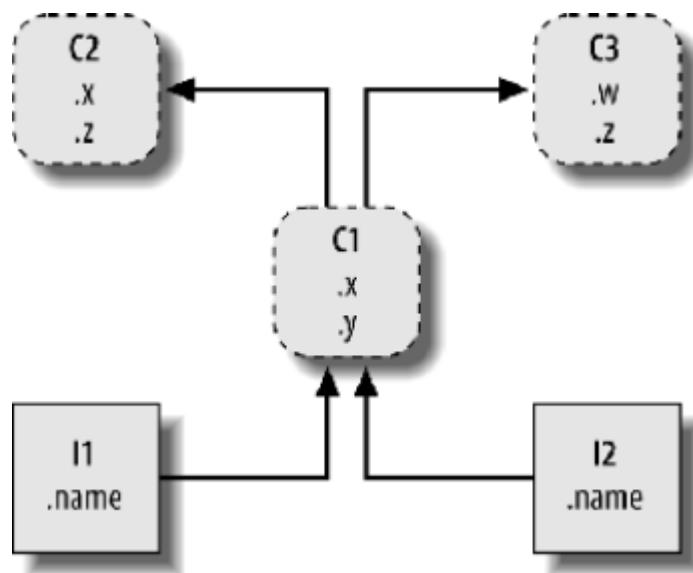
Wyrażenie to wykorzystywaliśmy w całej książce w celu uzyskania dostępu do atrybutów modułów czy wywołania metod obiektów. Kiedy jednak zastosujemy je do obiektu pochodzącego z instrukcji `class`, uruchamia ono w Pythonie *wyszukiwanie* — przeszukuje drzewo połączonych obiektów, szukając pierwszego wystąpienia atrybutu, jakie może odnaleźć. Kiedy w grę wchodzą klasy, to wyrażenie Pythona można w efekcie przełożyć na następujące wyrażenie z języka naturalnego:

Znajdź pierwsze wystąpienie *atrybutu*, szukając w *obiekcie*, a następnie we wszystkich klasach powyżej niego, od dołu do góry i od lewej strony do prawej.

Innymi słowy, operacja pobrania atrybutu jest po prostu przeszukiwaniem drzewa. Pojęcie *dziedziczenia* jest stosowane, ponieważ obiekty znajdujące się niżej w drzewie dziedziczą atrybuty dołączone do obiektów umieszczonych w tym drzewie wyżej. W miarę postępowania

wyszukiwania od dołu do góry — w pewnym sensie obiekty połączone w drzewo są sumą wszystkich atrybutów zdefiniowanych we wszystkich rodzicach, aż do samej góry drzewa.

W Pythonie wszystko to jest bardzo dosłowne. W kodzie naprawdę budujemy drzewa połączonych obiektów, a Python rzeczywiście wspina się w górę tego drzewa w trakcie wyszukiwania atrybutów w czasie wykonywania — za każdym razem, gdy użyjemy wyrażenia `obiekt.atrybut`. Aby to lepiej zilustrować, na rysunku 26.1 przedstawiono przykład jednego z takich drzew.



Rysunek 26.1. Drzewo klasy z dwomainstancjami na dole (I1 oraz I2), klasą nad nimi (C1) i dwoma klasami nadnadrzędnymi na górze (C2 oraz C3). Wszystkie te obiekty są przestrzeniami nazw (pakietami zmiennych), a dziedziczenie jest po prostu przeszukiwaniem drzewa od dołu do góry w celu odnalezienia najniższego wystąpienia nazwy atrybutu. Kod sugeruje kształt takich drzew

Na rysunku widoczne jest drzewo z pięcioma obiektami podpisanymi za pomocą zmiennych; do każdego z nich dołączono atrybuty, a drzewo jest gotowe do przeszukania. Drzewo łączy razem trzy *obiekty klas* (owale C1, C2 oraz C3) i dwa *obiekty instancji* (prostokąty I1 oraz I2) w drzewo wyszukiwania dziedziczenia. Warto zauważyć, że w modelu obiektów Pythona klasy oraz instancje z nich wygenerowane są dwoma odrębnymi typami obiektów.

Klasy

Służą jako fabryki instancji. Ich atrybuty udostępniają zachowanie — dane oraz funkcje — dziedziczone przez wszystkie instancje z nich wygenerowane (na przykład funkcję obliczającą wynagrodzenie pracownika na podstawie jego płacy oraz przepracowanych godzin).

Instancje

Reprezentują konkretne elementy w domenie programu. Ich atrybuty zapisują dane różniące się dla określonych obiektów (na przykład numer PESEL określonego pracownika).

W kategoriach drzew wyszukiwania instancja dziedziczy atrybuty po klasie, natomiast klasa dziedziczy atrybuty po wszystkich klasach znajdujących się nad nią w drzewie.

Na rysunku 26.1 możemy jeszcze podzielić owale zgodnie z ich pozycją względną w drzewie. Zazwyczaj nazywamy klasy znajdujące się wyżej w drzewie (jak C2 czy C3) *klasami nadzędnymi* lub *nadklasami*. Klasy umieszczone niżej w drzewie (jak C1) są znane jako *klasy podrzędne* lub *podklasy*. Te pojęcia odnoszą się do względnych pozycji oraz ról w drzewie. Klasa nadzędna udostępniają zachowanie współdzielone przez wszystkie ich podklasy, jednak ponieważ wyszukiwanie następuje od dołu do góry, podklasy mogą przesłaniać zachowanie zdefiniowane w klasach nadzędnych, redefiniując zmienne z klas nadzędnych niżej w drzewie^[1].

Ponieważ kilka ostatnich zdań stanowi tak naprawdę sedno dostosowywania oprogramowania do własnych potrzeb w programowaniu zorientowanym obiektowo, rozwińmy nieco przedstawione w nich koncepcje. Założymy, że budujemy drzewo z rysunku 26.1 i w kodzie umieszczamy poniższe wyrażenie.

I2.w

Ten kod natychmiast wywołuje dziedziczenie. Ponieważ jest to wyrażenie typu *obiekt.atrybut*, wywołuje ono przeszukiwanie drzewa z rysunku 26.1. Python będzie szukał atrybutu w w obiekcie I2 oraz powyżej niego. Mówiąc dokładniej, przeszuka połączone obiekty w następującej kolejności:

I2, C1, C2, C3

i zatrzyma się na pierwszym dołączonym atrybutem w, jaki potrafi znaleźć (lub zwróci błąd, jeśli nic nie zostanie odnalezione). W tym przypadku atrybut w zostanie odszukany dopiero w klasie C3, ponieważ pojawia się jedynie w tym obiekcie. Innymi słowy, I2.w okazuje się C3.w ze względu na wyszukiwanie automatyczne. W kategoriach programowania zorientowanego obiektowo I2 dziedziczy atrybut w po C3.

W rezultacie dwie instancje dziedziczą po swoich klasach cztery atrybuty — w, x, y oraz z. Inne referencje do atrybutów podążą innymi ścieżkami w drzewie. Na przykład:

- Wyrażenia I1.x oraz I2.x odnajdują atrybut x w klasie C1 i zatrzymują się, ponieważ C1 jest niższe od C2.
- Wyrażenia I1.y oraz I2.y odnajdują atrybut y w klasie C1, ponieważ jest to jedyne miejsce, w którym się on pojawia.
- Wyrażenia I1.z oraz I2.z odnajdują atrybut z w klasie C2, ponieważ C2 znajduje się bardziej na lewo od C3.
- Wyrażenie I2.name odnajduje atrybut name bez przechodzenia w góre drzewa.

Warto prześledzić te wyrażenia na drzewie z rysunku 26.1, żeby poczuć, w jaki sposób w Pythonie działa wyszukiwanie dziedziczenia.

Pierwszy element powyższej listy jest chyba najważniejszym, na jaki należy zwrócić uwagę. Ponieważ C1 redefiniuje atrybut x niżej w drzewie, w rezultacie *zastępuje* wersję znajdującą się wyżej w C2. Jak zobaczymy za moment, redefiniując i zastępując atrybut, C1 dostosowuje to, co dziedziczy po klasie nadzędnej, do własnych potrzeb.

Klasy a instancje

Choć w modelu Pythona są to,ściśle rzecz biorąc, dwa osobne typy obiektów, klasy oraz instancje umieszczone w drzewach są prawie identyczne. Głównym celem każdego typu jest służenie jako kolejny rodzaj *przestrzeni nazw* — pakietu zmiennych oraz miejsca, do którego można dołączać atrybuty. Jeśli klasy i instancje przypominają moduły, to dobrze — tak powinno być. Obiekty w drzewach klas zawierają również automatycznie przeszukiwane łącza do innych obiektów przestrzeni nazw, natomiast klasy odpowiadają instrukcjom, nie całym plikom.

Podstawowa różnica pomiędzy klasami i instancjami polega na tym, że klasy są swego rodzaju *fabryką* generującą instancje. W prawdziwej aplikacji możemy na przykład mieć klasę Employee

definiującą, co oznacza bycie pracownikiem. Z tej klasy generujemy konkretne instancje pracowników. Jest to kolejna różnica pomiędzy klasami a modułami — w pamięci zawsze mamy tylko jedną instancję określonego modułu (dlatego do pobrania aktualionego kodu trzeba moduł przeładować), natomiast w przypadku klas możemy utworzyć dowolną liczbę instancji.

Z punktu widzenia działania do klas zazwyczaj dołączone są funkcje (na przykład `computeSalary` obliczającą wynagrodzenie), a instancje będą zawierały bardziej podstawowe elementy danych wykorzystywane przez funkcje klas (na przykład zmienną `hoursWorked` wskazującą liczbę przepracowanych godzin). Tak naprawdę model zorientowany obiektowo nie różni się szczególnie od klasycznego modelu przetwarzania danych z *programami i rekordami*. W przypadku programowania zorientowanego obiektowo instancje są jak rekordy z „dynam”, a klasy są „programami” służącymi do przetwarzania tych rekordów. W programowaniu zorientowanym obiektowo mamy jednak dodatkowo pojęcie hierarchii dziedziczenia, która obsługuje możliwość dostosowania programu do własnych potrzeb lepiej od poprzednich modeli.

Wywołania metod klas

W poprzednim podrozdziale widzieliśmy, jak referencja do atrybutu `I2.w` w naszym przykładowym drzewie klas została przetłumaczona na `C3.w` za pomocą procedury wyszukiwania dziedziczenia w Pythonie. Chyba równie ważne do zrozumienia jak dziedziczenie atrybutów jest to, co dzieje się, kiedy próbujemy wywołać *metody* klas (czyli funkcje dołączone do klas jako atrybuty).

Jeżeli referencja `I2.w` jest wywołaniem *funkcji*, oznacza tak naprawdę: „Wywołaj funkcję `C3.w`, tak by przetworzyła ona `I2`”. W rezultacie Python automatycznie odwzorowuje wywołanie `I2.w()` na wywołanie `C3.w(I2)`, przekazując instancję jako pierwszy argument do odziedziczonej funkcji.

Tak naprawdę za każdym razem, gdy w ten sposób wywołujemy funkcję dołączoną do klasy, zasugerowana zostaje instancja klasy. Ten domniemany podmiot czy kontekst jest częścią powodu, dla którego model ten nazywamy *zorientowanym obiektowo* — kiedy operacja jest wykonywana, zawsze istnieje obiekt podmiotu. W bardziej realistycznym przykładzie możemy wywołać przyznającą podwyżkę metodę o nazwie `giveRaise`, dołączoną jako atrybut do klasy `Employee`. Takie wywołanie nie ma znaczenia, o ile nie zostanie do niego dodany pracownik, któremu powinna zostać przyznana podwyżka.

Jak zobaczymy później, Python przekazuje domniemaną instancję do specjalnego pierwszego argumentu metody, zgodnie z przyjętą konwencją nazywanego `self`. Metody wykorzystują ten argument do przetwarzania podmiotu wywołania. Jak się niebawem dowiesz, metody można wywołać albo przez instancję (na przykład `aleksander.giveRaise()`), albo przez klasę (na przykład `Employee.giveRaise(aleksander)`) — w naszych skryptach będziemy używać obu sposobów wywołania. Wywołania te ilustrują również obie kluczowe koncepcje programowania zorientowanego obiektowo: aby uruchomić wywołanie metody `aleksander.giveRaise()`, Python wykonuje następujące operacje:

1. Lokalizuje metodę `giveRaise` obiektu `aleksander` poprzez wyszukiwanie metod dziedziczonych.
2. Przekazuje obiekt `aleksander` do zlokalizowanej metody `giveRaise` w specjalnym argumencie `self`.

Wywołując metodę `Employee.giveRaise(aleksander)`, po prostu wykonujesz oba kroki samodzielnie. Opisywany jest tutaj przypadek domyślny (Python ma jeszcze dodatkowe typy metod, które poznamy później), ale dotyczy on znacznej większości kodu zorientowanego obiektowo napisanego w tym języku. Aby zobaczyć, w jaki sposób metody otrzymują podmioty do przetwarzania, musimy przejść do przykładowego kodu.

Tworzenie drzew klas

Choć mówimy tutaj w kategoriach abstrakcyjnych, za tymi wszystkimi koncepcjami stoi namacalny kod. Drzewa oraz ich obiekty konstruujemy za pomocą instrukcji `class` oraz wywołań klas, które bardziej szczegółowo omówimy później. W skrócie:

- Każda instrukcja `class` generuje nowy obiekt klasy.
- Za każdym razem gdy klasa jest wywoływana, generuje ona nowy obiekt instancji.
- Instancje są automatycznie połączone z klasami, przez które zostały utworzone.
- Klasa jest automatycznie łączona ze swoimi klasami nadzędnymi zgodnie z kolejnością, w jakiej wymieniamy je w nawiasach w wierszu nagłówka instrukcji `class`; kolejność nazw klas od lewej do prawej strony odpowiada ich hierarchii w drzewie.

Na przykład, aby zbudować drzewo z rysunku 26.1, możemy wykonać kod pokazany poniżej. Podobnie jak jest w przypadku definicji funkcji, klasy są zwykle kodowane w plikach modułów i uruchamiane podczas importu (aby zaoszczędzić nieco miejsca, pominąłem szczegóły instrukcji `class`).

```
class C2: ...                                # Utworzenie obiektów klas
(owale na rysunku)

class C3: ...

class C1(C2, C3): ...                         # Połączona z klasami
nadzędnymi (w tej kolejności)

I1 = C1()                                     # Utworzenie obiektów instancji
(prostokąty na rysunku)

I2 = C1()                                     # Połączenie z ich klasami
```

W powyższym kodzie budujemy trzy obiekty klas, wykonując trzy instrukcje `class`, a także tworzymy dwa obiekty instancji, wywołując dwukrotnie `C1`, tak, jakby była to funkcja. Instancje pamiętają, z których klas zostały utworzone, natomiast klasa `C1` pamięta swoje klasę nadzędne.

Z technicznego punktu widzenia przykład ten wykorzystuje coś o nazwie *dziedziczenia wielokrotnego* (ang. *multiple inheritance*), co oznacza po prostu, że klasa może mieć więcej niż jedną klasę nadzelną w drzewie klas — jest to bardzo użyteczna technika, kiedy chcesz połączyć wiele narzędzi. W Pythonie, jeżeli w instrukcji `class` w nawiasach wymieniona jest większa liczba klas nadzędnych (jak w przypadku `C1` w kodzie wyżej), ich uporządkowanie od lewej do prawej strony określa kolejność, w jakiej klasy te będą przeszukiwane pod kątem dziedziczonych atrybutów. Domyślnie używana będzie nazwa wymieniona jako pierwsza od lewej strony, chociaż zawsze możesz wybrać dowolną nazwę, wybierając ją z klas, w której rezyduje (np. `C3.z`).

Ze względu na sposób działania wyszukiwania dziedziczonych komponentów obiekt, do którego dołączymy atrybut, okazuje się mieć kluczowe znaczenie — określa zasięg zmiennej. Atrybuty dołączone do instancji odnoszą się tylko do tych instancji, jednak atrybuty dołączone do klas są współdzielone przez wszystkie ich podklasy oraz instancje. Później szczegółowo przestudiujemy kod załączający atrybuty do takich obiektów. Dowiemy się, że:

- atrybuty są zwykle dołączane do klas przez przypisania dokonane na najwyższym poziomie w blokach instrukcji `class`, a nie zagnieżdżone w instrukcjach `def` tworzących funkcje;
- atrybuty są zazwyczaj dołączane do instancji przez przypisanie do specjalnego argumentu przekazywanego do funkcji wewnętrznej klasy i noszącego nazwę `self`.

Na przykład klasy udostępniają zachowanie swoim instancjom za pomocą funkcji utworzonych przez instrukcje `def` umieszczone w instrukcjach `class`. Ponieważ takie zagnieżdżone

instrukcje `def` przypisują nazwy wewnętrz klasy, w rezultacie dołączają do obiektu klasy atrybuty, które zostają odziedziczone przez wszystkie instancje i klasy podległe.

Z punktu widzenia składni w instrukcjach `def` w tym kontekście nie ma niczego nadzwyczajnego. Z operacyjnego punktu widzenia, kiedy instrukcja `def` pojawia się wewnątrz `class`, zazwyczaj jest nazywana *metodą* i automatycznie otrzymuje specjalny pierwszy argument, zgodnie z przyjętą konwencją nazywany `self`, udostępniający uchwyt zwrotny do przetwarzanej instancji. Dowolne inne wartości przekazywane do metody są umieszczane w kolejnych argumentach po `self` (w naszym przykładzie jest to na przykład argument `who`).^[2]

Ponieważ klasy są fabrykami tworzącymi wiele instancji, ich metody zazwyczaj używają automatycznie przekazanego argumentu `self` za każdym razem, gdy potrzebują pobrać lub ustawić atrybuty określonej instancji przetwarzanej przez wywołanie metody. W poprzednim kodzie argument `self` wykorzystano do przechowania nazwy w jednej z dwóch instancji.

Podobnie jak jest w przypadku prostych zmiennych, atrybuty klas oraz instancji nie są deklarowane z wyprzedzeniem, a zamiast tego zaczynają istnieć w momencie pierwszego przypisania do nich wartości. Kiedy metoda przypisuje coś do atrybutu `self`, tworzy bądź modyfikuje atrybut w instancji na dole drzewa klas (np. w jednym z prostokątów na rysunku 26.1), ponieważ `self` automatycznie odnosi się do przetwarzanej instancji — czyli podmiotu wywołania.

Tak naprawdę, ponieważ wszystkie obiekty drzewa klas są po prostu obiektami przestrzeni nazw, możemy pobrać lub ustawić każdy z ich atrybutów poprzez użycie odpowiednich nazw. Na przykład użycie wywołania `C1.setname` jest tak samo poprawne jak `I1.setname`, o ile zmienne `C1` i `I1` znajdują się w zasięgach naszego kodu.

Przeciążanie operatorów

W obecnej formie nasza klasa C1 nie dołącza atrybutu name do instancji, dopóki nie zostanie wywołana metoda setname. Tak naprawdę odwołanie się do I1.name przed wywołaniem metody I1.setname powoduje zwrócenie błędu spowodowanego niezdefiniowaną zmienną. Jeżeli chcemy zagwarantować w klasie, że atrybut taki jak name jest zawsze ustawiony w instancjach, zazwyczaj wypełniamy ten atrybut w czasie tworzenia, jak w poniższym przykładzie.

```
class C2: ...                                # Utworzenie obiektów klasy  
nadrzednej
```

```

class C3: ...
class C1(C2, C3):
    def __init__(self, who):
        self.name = who
# Ustawia name przy utworzeniu
# self to albo I1, albo I2
I1 = C1('amadeusz')
# Ustawia I1.name na 'amadeusz'
I2 = C1('aleksander')
# Ustawia I2.name na
'aleksander'
# Wyświetla 'amadeusz'
print(I1.name)

```

Po utworzeniu takiego kodu lub jego odziedziczeniu Python automatycznie wywołuje metodę o nazwie `__init__` za każdym razem, gdy generowana jest instancja klasy. Nowa instancja przekazywana jest do argumentu `self` metody `__init__` tak, jak zawsze, a wartości podane w nawiasach w wywołaniu klasy trafiają do argumentów od drugiego w górę. W rezultacie inicjalizujemy instancję w czasie, gdy jest ona tworzona, bez wymagania dodatkowych wywołań metody.

Metoda `__init__` znana jest jako *konstruktor* ze względu na czas wykonywania. To najczęściej wykorzystywany reprezentant większej klasy metod znanych jako *metody przeciążania operatorów*, które omówimy bardziej szczegółowo w kolejnych rozdziałach. Takie metody są dziedziczone w drzewach klas tak, jak zawsze i na początku oraz końcu mają podwójne znaki `_`, co pozwala je odróżnić od innych metod. Python wykonuje je automatycznie, kiedy instancje je obsługujące pojawiają się w odpowiadających im operacjach. Są w większości alternatywą dla wykonywania prostych wywołań metod. Są również opcjonalne — jeżeli je pominiemy, operacje te nie są obsługiwane. Jeżeli metoda `__init__` nie istnieje, wywołanie klasy zwraca pustą instancję, bez jej inicjalizowania.

Na przykład, aby zaimplementować część wspólną zbiorów, klasa może albo udostępniać metodę o nazwie `intersect`, albo przeciągać operator wyrażenia `&` w taki sposób, aby udostępnić wymaganą logikę w nowej wersji metody o nazwie `__and__`. Ponieważ rozwiązanie z operatorem sprawia, że instancje wyglądają i zachowują się bardziej jak typy wbudowane, pozwala to klasom udostępniać konsekwentny i naturalny interfejs, a także zapewnia spójność z kodem oczekującym typu wbudowanego. Mimo to, poza konstruktorem `__init__`, który pojawia się w najbardziej realistycznych klasach, wiele programów może sobie lepiej radzić ze zwykłymi nazwanymi metodami, chyba że ich obiekty są podobne do wbudowanych. Metoda `giveRaise` może mieć sens dla obiektu `Employee`, ale już nowy operator `&` niekoniecznie.

Programowanie zorientowane obiektowo oparte jest na ponownym wykorzystaniu kodu

I właśnie to wraz z kilkoma szczegółami składniowymi składa się na większość zagadnień związanych z programowaniem zorientowanym obiektowo w Pythonie. Oczywiście w grę wchodzi coś więcej niż tylko dziedziczenie. Przeciążanie operatorów jest o wiele bardziej ogólne, niż to dotychczas opisałem — klasy mogą również udostępniać swoje własne implementacje operacji takich, jak indeksowanie, pobieranie atrybutów czy wyświetlanie. Przede wszystkim jednak programowanie zorientowane obiektowo polega na wyszukiwaniu atrybutów w drzewach z użyciem wspomnianego wcześniej specjalnego, pierwszego argumentu funkcji.

Dlaczego mielibyśmy interesować się budowaniem i przeszukiwaniem drzew obiektów? Choć zrozumienie tego wymaga nieco praktyki, to dobrze wykorzystywane klasy wspomagają *ponowne użycie kodu* w sposób niemożliwy do uzyskania za pomocą innych komponentów programu Pythona. W praktyce jest to jedna z ich głównych zalet i zastosowań. Dzięki klasom możemy tworzyć nowy kod, dostosowując istniejące oprogramowanie do własnych potrzeb,

zamiast albo modyfikować je w miejscu, albo rozpoczynać każdy nowy projekt od podstaw. Takie rozwiązanie okazuje się być potężnym paradygmatem w programowaniu praktycznym.

Na poziomie podstawowym klasy są tak naprawdę pakietami funkcji i innych zmiennych, podobnie jak moduły. Automatyczne wyszukiwanie dziedziczenia atrybutów będące częścią klas umożliwia dostosowanie oprogramowania do własnych potrzeb wykraczające poza to, co możemy uzyskać za pomocą modułów oraz funkcji. Co więcej, klasy stanowią naturalną strukturę kodu, która pakuje i lokalizuje logikę oraz nazwy, co znakomicie ułatwia debugowanie kodu.

Przykładowo, ponieważ metody są po prostu funkcjami ze specjalnym pierwszym argumentem, możemy naśladować ich zachowanie, ręcznie przekazując obiekty, jakie mają być przetwarzane, do prostych funkcji. Udział metod w dziedziczeniu klas pozwala nam jednak w naturalny sposób dostosowywać istniejące oprogramowanie do własnej sytuacji za pomocą tworzenia klas podrzędnych z nowymi definicjami metod zamiast modyfikowania istniejącego kodu w miejscu. Takie coś w przypadku modułów oraz funkcji nie jest możliwe.

Polimorfizm i klasy

Jako przykład rozważmy sytuację, w której otrzymaliśmy zadanie zaimplementowania aplikacji działającej na bazie danych pracowników. Jako programista Pythona zorientowany obiektowo możesz rozpocząć od napisania ogólnej klasy nadzędnej definiującej domyślne zachowania wspólne dla wszystkich typów pracowników naszej organizacji.

```
class Employee:                                # Ogólna klasa nadzędna
    def computeSalary(self): ...                # Wspólne lub domyślne
    zachowanie
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

Po zaimplementowaniu tego ogólnego zachowania w kodzie możemy je teraz wyspecjalizować dla każdego konkretnego typu pracownika w celu odzwierciedlenia tego, jak poszczególne typy odbiegają od normy. Możemy zatem utworzyć klasy podrzędne dostosowujące te fragmenty zachowania, które są różne dla różnych typów pracowników. Pozostałe wzorce zachowania pracowników zostaną odziedziczone po klasie ogólnej. Jeżeli na przykład inżynierowie mają unikalne reguły obliczania wynagrodzenia (na przykład nie korzystają ze stawki godzinowej), wystarczy zastąpić jedną metodę w klasie podrzędnej.

```
class Engineer(Employee):                      # Wyspecjalizowana klasa
    podrzędna
    def computeSalary(self): ...                # Tutaj wstawiamy
    niestandardowy, własny kod metody
```

Ponieważ ta wersja metody `computeSalary` pojawia się niżej w drzewie klas, zastąpi (przesłoni) ogólną wersję z klasy `Employee`. Możemy następnie utworzyć podobne instancje dla klas pracowników, do których należą konkretne osoby, tak by otrzymać poprawne zachowanie.

```
amadeusz = Employee()                        # Zachowanie domyślne
agata = Employee()                            # Zachowanie domyślne
aleksander = Engineer()                      # Własne obliczenie
wynagrodzenia
```

Warto zauważyć, że możemy utworzyć instancje dowolnej klasy drzewa — nie tylko klas znajdujących się na dole. Klasa, której instancję tworzymy, określa poziom, na jakim rozpoczęcie się wyszukiwanie atrybutów, a tym samym to, które wersje metod zostaną użyte.

W rezultacie wymienione trzy instancje obiektów mogą zostać osadzone w jakimś większym obiekcie spełniającym rolę kontenera (na przykład liście lub w instancji innej klasy), który reprezentuje dział czy nawet całą firmę, wykorzystując do tego koncepcję kompozycji wspomnianą na początku rozdziału. Kiedy później będziemy pytać o zarobki tych pracowników, zostaną one obliczone zgodnie z klasami, w których utworzone zostały poszczególne obiekty, ze względu na reguły wyszukiwania dziedziczonych komponentów.

```
company = [amadeusz, agata, aleksander]      # Obiekt kompozytowy
for emp in company:
    print(emp.computeSalary())                # Wykonanie odpowiedniej wersji
                                                # obiektu (domyślnej lub dostosowanej)
```

Jest to kolejny przykład koncepcji *polimorfizmu* wprowadzonej w rozdziale 4. i rozszerzonej w rozdziale 16. Warto przypomnieć, że polimorfizm oznacza, iż znaczenie operacji uzależnione jest od obiektu, na którym się ona odbywa, czyli inaczej mówiąc, kod programu nie powinien zajmować się tym, czym jest dany obiekt, a tylko tym, co on robi. W naszym przykładzie metoda `computeSalary` przed wywołaniem lokalizowana jest za pomocą wyszukiwania dziedziczonych komponentów w każdym obiekcie. W rezultacie zawsze automatycznie zostanie uruchomiona poprawna wersja metody dla przetwarzanego obiektu. Aby przekonać się, dlaczego tak się dzieje, powinieneś uważnie przeanalizować kod tego przykładu [\[3\]](#).

W innych aplikacjach polimorfizm może również służyć do ukrywania (np. za pomocą *hermetyzacji*) różnic między interfejsami. Przykładowo program przetwarzający strumienie danych może zostać zapisany w kodzie w taki sposób, by oczekiwał obiektów z metodami wejścia i wyjścia, bez troszczenia się o to, co te metody tak naprawdę robią.

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Przekazując instancje klas podrzędnych specjalizujących wymagane interfejsy metod `read` oraz `write` dla różnych źródeł danych, możemy wykorzystać funkcję `processor` ponownie dla dowolnego źródła danych, z jakiego będziemy musieli korzystać — teraz i w przyszłości.

```
class Reader:
    def read(self): ...                                # Domyślne zachowanie oraz
                                                       # narzędzia
    def other(self): ...
class FileReader(Reader):
    def read(self): ...                                # Odczytanie z lokalnego pliku
class SocketReader(Reader):
    def read(self): ...                                # Odczytanie z gniazda
                                                       # sieciowego
...
processor(FileReader(...), Converter, FileWriter(...))
```

```
processor(SocketReader(...), Converter, TapeWriter(...))  
processor(FtpReader(...), Converter, XmlWriter(...))
```

Co więcej, ponieważ wewnętrzne implementacje metod `read` oraz `write` zostały umieszczone w osobnych lokalizacjach, można je modyfikować bez wpływu na wykorzystujący je kod. Tak naprawdę sama funkcja `processor` może być klasą, co pozwoli na wypełnienie logiki konwersji w `converter` za pomocą dziedziczenia, a także umożliwi osadzenie kodu odczytującego i zapisującego według kompozycji (w dalszej części książki zobaczymy, jak to działa).

Programowanie przez dostosowanie

Kiedy przyzwyczajmy się do programowania w ten sposób (czyli dostosowywania oprogramowania do własnych potrzeb), zobaczymy, że gdy przyjdzie nam napisać nowy program, okaże się, iż duża część pracy została już wykonana — nasze zadanie będzie się w dużej mierze sprowadzało do połączenia ze sobą istniejących klas nadrzędnych implementujących zachowanie wymagane przez program. Przykładowo ktoś mógł już napisać klasy `Employee`, `Reader` oraz `Writer` z tego przykładu w celu zastosowania ich w zupełnie innym programie. Jeżeli tak, cały kod utworzony przez tę osobę otrzymujemy „za darmo”.

W wielu dziedzinach zastosowań można pobrać lub kupić gotowe kolekcje klas nadrzędnych, znanych jako *platformy* (ang. *framework*) i implementujących często wykonywane zadania programistyczne w postaci klas gotowych do dołączenia do naszych aplikacji. Takie platformy mogą udostępniać interfejsy do baz danych, protokoły testowania czy zestawy narzędzi graficznego interfejsu użytkownika. Dzięki takim platformom wystarczy często utworzyć klasę podzieloną dodającą oczekiwany metodę czy nawet kilka metod, natomiast większość pracy zostanie wykonana przez klasy nadrzędne znajdujące się wyżej w drzewie. Programowanie w świecie zorientowanym obiektowo polega na łączeniu i specjalizacji gotowego i sprawdzonego kodu za pomocą pisania własnych klas podzielonych.

Oczywiście nauczenie się, jak można wykorzystywać klasy w taki sposób w celu uzyskania podobnego, zorientowanego obiektowo efektu, z pewnością zajmuje trochę czasu. W praktyce programowanie zorientowane obiektowo obejmuje również sporą dawkę projektowania wymaganą do pełnego zrozumienia zalet ponownego wykorzystywania kodu dzięki klasom. W tym celu programiści zaczęli katalogować najczęściej wykorzystywane struktury programowania zorientowanego obiektowo, znane jako *wzorce projektowe* (ang. *design patterns*), które mogą pomóc nam w projektowaniu. Sam kod, jaki piszemy w Pythonie w programowaniu zorientowanym obiektowo, jest tak prosty, że nie powinien stanowić dla nas żadnej dodatkowej przeszkody. Żeby to jednak zobaczyć, będziemy musieli przejść do rozdziału 27.

Podsumowanie rozdziału

W niniejszym rozdziale przyjrzaliśmy się klasom oraz programowaniu zorientowanemu obiektowo na poziomie abstrakcji, próbując zyskać szerszą perspektywę, zanim przejdziemy do szczegółów składni. Jak sam mogłeś się przekonać, programowanie zorientowane obiektowo w dużej mierze opiera się na argumencie o nazwie `self` i wyszukiwaniu atrybutów w drzewach połączonych obiektów, co nazywamy dziedziczeniem. Obiekty znajdujące się na dole drzewa dziedziczą atrybuty po obiektach znajdujących się wyżej — pozwala nam to programować za pomocą dostosowywania kodu do własnych potrzeb, a nie modyfikowania go czy rozpoczęmania od nowa. Kiedy dobrze go wykorzystujesz, taki model programowania może radykalnie skrócić czas poświęcony na tworzenie kodu.

W kolejnym rozdziale zaczniemy wypełniać brakujące fragmenty szczegółów dotyczących kodu, stojących za zarysowaną tutaj perspektywą. W miarę zagłębiania się w klasy Pythona należy pamiętać o tym, że model programowania zorientowanego obiektowo w Pythonie jest bardzo

prosty — jak powiedzieliśmy wcześniej, tak naprawdę sprowadza się do wyszukiwania atrybutów w drzewach obiektów i specjalnego argumentu funkcji. Zanim jednak przejdziemy dalej, czas na krótki quiz sprawdzający przedstawione dotychczas informacje.

Sprawdź swoją wiedzę — quiz

1. Co jest głównym celem programowania zorientowanego obiektowo w Pythonie?
2. Gdzie wyszukiwanie dziedziczonych komponentów poszukuje atrybutów?
3. Jaka jest różnica pomiędzy obiektem klasy a obiektem instancji?
4. Dlaczego pierwszy argument funkcji metody klasy jest specjalny?
5. Do czego wykorzystywana jest metoda `__init__`?
6. W jaki sposób tworzymy instancję klasy?
7. W jaki sposób tworzymy klasę?
8. W jaki sposób określamy klasę nadzczną danej klasy?

Sprawdź swoją wiedzę — odpowiedzi

1. Programowanie zorientowane obiektowo opiera się na ponownym wykorzystywaniu kodu — kod poddawany jest faktoryzacji w celu zminimalizowania jego powtarzalności, a programuje się, dostosowując istniejący kod do własnych potrzeb, zamiast modyfikować go w miejscu czy pisać od nowa.
2. Wyszukiwanie dziedziczonych komponentów poszukuje atrybutów najpierw w obiekcie instancji, później w klasie, w której powstała ta instancja, a następnie we wszystkich klasach nadzędnych, domyślnie przechodząc przez drzewa obiektów od dołu do góry i od lewej strony do prawej. Wyszukiwanie zatrzymuje się w pierwszym miejscu, w jakim odnaleziony zostaje atrybut. Ponieważ najniższa wersja zmiennej znaleziona po drodze wygrywa, hierarchie klas w naturalny sposób obsługują dostosowywanie kodu do własnych potrzeb za pomocą rozszerzania go w nowych klasach podrzędnych.
3. Zarówno obiekty klas, jak i instancje obiektów są przestrzeniami nazw (pakietami zmiennych, które dostępne są jako atrybuty). Podstawowa różnica między nimi polega na tym, że klasy są rodzajem fabryki służącej do tworzenia wielu instancji. Klasa obsługuje również dziedziczone przez instancje metody przeciążania operatorów i traktują funkcje zagnieżdżone wewnętrz nich jako specjalne metody służące do przetwarzania instancji.
4. Pierwszy argument w metodzie klasy jest specjalny, ponieważ zawsze otrzymuje on instancję obiektu będącego domniemanym podmiotem wywołania metody. Zazwyczaj nosi on nazwę `self`, zgodnie z przyjętą konwencją. Ponieważ funkcje metod zawsze zawierają ten kontekst domniemanego podmiotu, mówimy, że są „zorientowane obiektowo”, czyli zaprojektowane tak, by przetwarzać lub modyfikować obiekty.
5. Jeżeli metoda `__init__` jest zapisana w kodzie lub dziedziczona w klasie, Python wywołuje ją automatycznie za każdym razem, gdy tworzona jest instancja tej klasy.

Jest ona znana jako metoda konstruktora i jest przekazywana do każdej nowej instancji w sposób niejawnym, wraz z dowolnymi argumentami przekazanymi do nazwy klasy w sposób jawnym. Metoda `__init__` jest również najczęściej wykorzystywana metodą przeciążania operatorów. Jeżeli nie występuje ona w klasie, instancje zaczynają swoje istnienie jako puste przestrzenie nazw.

6. Instancję klasy tworzymy, wywołując nazwę klasy, tak jakby była ona funkcją. Argumenty przekazane do nazwy klasy pojawiają się jako argumenty na pozycjach drugiej i kolejnych w metodzie konstruktora `__init__`. Nowa instancja pamięta klasę, z której została utworzona, ze względu na dziedziczenie.
7. Klasę tworzymy, wykonując instrukcję `class`. Podobnie do definicji funkcji, instrukcje te normalnie wykonywane są, kiedy importowany jest zawierający je plik modułu (więcej informacji na ten temat w kolejnym rozdziale).
8. Klasy nadrzędne określamy, wymieniając je w nawiasach w instrukcji `class`, zaraz po nazwie nowej klasy. Uporządkowanie klas w nawiasach od lewej do prawej strony określa kolejność wyszukiwania dziedziczenia w drzewie klas.

[1] W innej literaturze można się również czasami spotkać z pojęciami *klasa podstawowa* oraz *klasa pochodna*, które służą do opisywania, odpowiednio, klas nadrzędnego i podrzędnego. Użytkownicy Pythona używają zwykle tego drugiego zestawu określeń i tego też będziemy trzymać się w tej książce.

[2] Jeżeli kiedykolwiek używałeś języków C++ lub Java, z pewnością rozpoznajesz, że argument `self` w Pythonie jest tym samym co wskaźnik `this`, jednak argument `self` z Pythona jest zawsze jawnym zarówno w nagłówkach, jak i ciałach metod, tak by dostęp do atrybutów był bardziej oczywisty.

[3] Warto zauważyć, że lista `company` w tym przykładzie mogłaby być bazą danych, jeżeli byłaby zapisana w pliku z użyciem modułu `pickle` Pythona, wprowadzonego w rozdziale 9. — w ten sposób moglibyśmy uzyskać trwałą bazę danych pracowników. Python zawiera również moduł `shelve`, który pozwala na przechowywanie poddanych serializacji instancji klas w systemie plików wykorzystującym dostęp według klucza; wdrożymy go w rozdziale 28.

Rozdział 27. Podstawy tworzenia klas

Skoro omówiliśmy już abstrakcyjną część programowania zorientowanego obiektowo, czas zaprezentować, jak przekłada się ona na prawdziwy kod. W niniejszym rozdziale zacznemy omawiać szczegóły składni stojącej za modelem klas w Pythonie.

Osobom, które nigdy w przeszłości nie spotkały się z programowaniem zorientowanym obiektowo, klasy mogą się wydawać nieco skomplikowane, zwłaszcza jeżeli się je próbuje przyswoić w jednym kawałku. Aby tworzenie klas było łatwiejsze do zrozumienia, omówienie programowania zorientowanego obiektowo rozpoczęmy w tym rozdziale od przyjrzenia się działaniu podstawowych klas. Przedstawione tutaj podstawy rozszerzymy w kolejnych rozdziałach tej części książki. W swojej podstawowej postaci klasy Pythona są łatwe do zrozumienia.

Tak naprawdę klasy mają trzy podstawowe cechy wyróżniające. Na najbardziej podstawowym poziomie są po prostu przestrzeniami nazw, podobnie jak moduły omówione w piątej części książki. Jednak w przeciwieństwie do modułów klasy obsługują również generowanie wielu obiektów, dziedziczenie przestrzeni nazw oraz przeciążanie operatorów. Zaczniemy nasze omawianie klas od przyjrzenia się każdej z tych trzech cech wyróżniających.

Klasy generują wiele obiektów instancji

Aby zrozumieć, jak działa koncepcja wielu obiektów, musisz najpierw zrozumieć, że w modelu programowania zorientowanego obiektowo w Pythonie istnieją dwa rodzaje obiektów — obiekty *klas* oraz obiekty *instancji*. Obiekty klas udostępniają domyślne zachowanie i służą jako fabryki obiektów instancji. Obiekty instancji są prawdziwymi obiektami przetwarzanymi przez programy — każdy jest osobną przestrzenią nazw, jednak jednocześnie dziedziczy (to znaczy automatycznie ma dostęp) zmienne z klasy, w której został utworzony. Obiekty klas pochodzą z instrukcji, a obiekty instancji — z wywołań. Za każdym razem gdy wywołujemy klasę, otrzymujemy nową instancję tej klasy.

Taka koncepcja generowania obiektów jest całkowicie różna od wszystkich konstrukcji programu, które dotychczas widzieliśmy. W rezultacie klasy są tak naprawdę *fabrykami* generującymi wiele instancji. W przeciwieństwie do tego tylko jedna kopia modułu jest importowana do jednego programu. W rzeczywistości dlatego instrukcja `reload` działa tak, jak działa, aktualizując w miejscu pojedynczą instancję współdzielonego obiektu. Dzięki klasom każda instancja może mieć własne, niezależne dane, obsługujące wiele wersji obiektu modelowanego przez daną klasę.

W tej roli instancje klas są podobne do funkcji domykających przechowujących stan między wywołaniami (nazywanych też funkcjami fabrykującymi), które omawialiśmy w rozdziale 17., ale jest to naturalna część modelu klas, a stany w klasach są atrybutami jawnymi, w przeciwieństwie do niejawnych odwołań w zasięgach. Co więcej, jest to tylko część tego, co robią klasy — obsługują także dostosowywanie poprzez dziedziczenie, przeciążanie operatorów i wiele zachowań za pośrednictwem metod. Ogólnie rzecz biorąc, klasy są bardziej kompletnym narzędziem programistycznym, chociaż programowanie zorientowane obiektowo i

programowanie funkcyjne nie wykluczają się wzajemnie. Możemy łączyć je za pomocą narzędzi funkcyjnych w metodach, kodując metody, które same są generatorami, tworząc iteratory zdefiniowane przez użytkownika (jak zobaczymy w rozdziale 30.) i tak dalej.

Poniżej znajduje się krótkie streszczenie podstaw programowania zorientowanego obiektowo w Pythonie w kontekście jego dwóch typów obiektów. Jak zobaczymy, klasy Pythona w wielu aspektach są podobne do instrukcji `def` oraz modułów, jednak mogą się dość znacznie różnić od tego, co możemy znać z innych języków programowania.

Obiekty klas udostępniają zachowania domyślne

Kiedy wykonujemy instrukcję `class`, otrzymujemy obiekt klasy. Poniżej znajduje się lista najważniejszych właściwości klas Pythona.

- **Instrukcja `class` tworzy obiekt klasy i przypisuje mu nazwę.** Podobnie jak instrukcja `def`, instrukcja `class` jest w Pythonie instrukcją wykonywalną. Kiedy interpreter Pythona dotrze do niej i ją wykona, generuje nowy obiekt klasy i przypisuje go do nazwy podanej w nagłówku instrukcji. Podobnie jak `def`, instrukcja `class` zazwyczaj wykonywana jest przy pierwszym importowaniu zawierającego ją pliku.
- **Przypisania wewnętrz instrukcji `class` tworzą atrybuty klasy.** Podobnie jak w plikach modułów, przypisania na najwyższym poziomie wewnętrz instrukcji `class` (nieosadzone wewnętrz `def`) generują atrybuty w obiekcie klasy. Z technicznego punktu widzenia zakres instrukcji `class` staje się przestrzenią nazw atrybutów obiektu, podobnie jak ma to miejsce w przypadku zakresu globalnego modułu. Po wykonaniu instrukcji `class` dostęp do atrybutów można uzyskać za pomocą składni kwalifikującej `obiekt.atrybut`.
- **Atrybuty klasy udostępniają stan obiektu oraz jego zachowanie.** Atrybuty obiektu klasy rejestrują informacje o stanie oraz zachowanie współdzielone przez wszystkie instancje utworzone z tej klasy. Instrukcje `def` funkcji zagnieżdżone wewnętrz instrukcji `class` generują metody, które przetwarzają instancje.

Obiekty instancji są rzeczywistymi elementami

Kiedy wywołujemy obiekt klasy, otrzymujemy obiekt instancji. Poniżej znajduje się przegląd najważniejszych właściwości instancji klas.

- **Wywołanie obiektu klasy w sposób podobny do wywołania funkcji tworzy nowy obiekt instancji.** Za każdym wywołaniem klasy tworzony i zwracany jest nowy obiekt instancji. Instancje reprezentują rzeczywiste elementy z domeną naszego programu.
- **Każdy obiekt instancji dziedziczy atrybuty klasy oraz otrzymuje własną przestrzeń nazw.** Obiekty instancji utworzone przez klasy są nowymi przestrzeniami nazw. Na początku są puste, ale dziedziczą atrybuty znajdujące się w obiektach klas, z których zostały wygenerowane.
- **Przypisania do atrybutów `self` w metodach tworzą atrybuty dla poszczególnych instancji.** Wewnętrz metod klasy pierwszy argument (o zgodnej z konwencją nazwie `self`) odnosi się do przetwarzanego obiektu instancji. Przypisania do atrybutów `self` tworzą lub modyfikują dane w instancji, a nie w klasie.

W rezultacie klasy definiują wspólne, współdzielone dane i zachowanie oraz generują instancje. Instancje odzwierciedlają konkretne komponenty aplikacji i rejestrują dane, które mogą się różnić w zależności od obiektu.

Pierwszy przykład

Zajmijmy się teraz pierwszym prawdziwym przykładem, który pokaże nam, jak koncepcje te działają w praktyce. Na początek zdefiniujmy klasę o nazwie `FirstClass`, wykonując instrukcję `class` Pythona w sesji interaktywnej.

```
>>> class FirstClass:                                # Zdefiniowanie obiektu klasy
    def setdata(self, value):                         # Zdefiniowanie metod klasy
        self.data = value                            # self to instancja
    def display(self):
        print(self.data)                            # self.data: dla instancji
```

Pracujemy tutaj w sesji interaktywnej, jednak zazwyczaj taką instrukcję wykonuje się, kiedy importowany jest plik, w którym została ona zapisana. Podobnie do funkcji utworzonych za pomocą instrukcji `def`, klasa ta nie istnieje, dopóki Python nie dotrze do tej instrukcji i nie wykona jej.

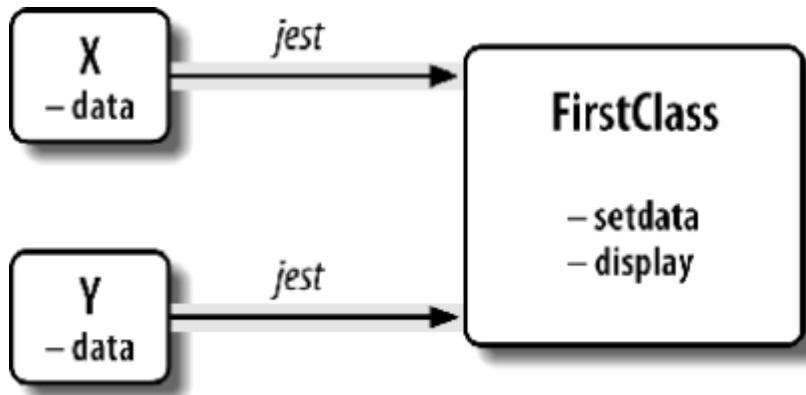
Tak jak wszystkie instrukcje złożone, `class` rozpoczyna się od wiersza nagłówka podającego jej nazwę, po której umieszczone jest ciało składające się z jednej lub większej liczby zagnieżdżonych i w zwykły sposób wciętych instrukcji. Tutaj zagnieżdżone instrukcje to `def`; definiują one funkcje implementujące zachowanie, jakie klasa chce eksportować.

Jak wiemy z czwartej części książki, instrukcja `def` jest tak naprawdę przypisaniem. W kodzie powyżej przypisuje obiekty funkcji do nazw `setdata` oraz `display` w zakresie instrukcji `class`, przez co generuje atrybuty dołączone do klas, noszące nazwy `FirstClass.setdata` oraz `FirstClass.display`. Tak naprawdę każda zmienna przypisana na najwyższym poziomie zagnieżdżonego bloku klasy staje się atrybutem tej klasy.

Funkcje znajdujące się wewnętrz klas zazwyczaj nazywane są *metodami*. W kodzie zapisywane są za pomocą normalnych instrukcji `def` i obsługują wszystko, czego dotychczas nauczyliśmy się o funkcjach (mogą na przykład mieć wartości domyślne, zwracać wartości po zakończeniu działania, zwracać wartości na żądanie i tak dalej). W metodach pierwszy argument automatycznie po wywołaniu otrzymuje domniemany obiekt instancji — podmiot wywołania. Aby zobaczyć, jak to działa, musimy utworzyć kilka instancji.

```
>>> x = FirstClass()                                # Utworzenie dwóch instancji
>>> y = FirstClass()                                # Każda jest nową przestrzenią
nazw
```

Wywołując klasę w ten sposób (należy zwrócić uwagę na nawiasy), generujemy obiekty instancji, które tak naprawdę są po prostu przestrzeniami nazw z dostępem do atrybutów klasy. Mówiąc dokładniej, w tym momencie mamy do dyspozycji trzy obiekty — dwie instancje oraz klasę. Tak naprawdę mamy trzy połączone przestrzenie nazw naszkicowane na rysunku 27.1. W terminologii programowania zorientowanego obiektowo mówimy, że `x` jest obiektem klasy `FirstClass`, podobnie jak `y` — oba obiekty dziedziczą nazwy przypisane do tej klasy.



Rysunek 27.1. Klasy oraz instancje są połączonymi obiektami przestrzeni nazw w drzewie klas, które jest przeszukiwane przez dziedziczenie. Atrybut „data” zostaje tutaj odnaleziony w instancjach, natomiast „setdata” oraz „display” w klasie nadzędnej

Dwie instancje są na początku puste, jednak mają łącza z powrotem do klasy, z której zostały wygenerowane. Jeżeli zapiszemy instancję wraz z nazwą atrybutu znajdującego się w obiekcie klasy w składni kwalifikującej (z kropką), Python pobierze tę nazwę z klasy za pomocą wyszukiwania dziedziczonych komponentów (o ile nie znajduje się ona również w instancji).

```

>>> x.setdata("Król Artur")          # Wywołanie metod: self to x
>>> y.setdata(3.14159)              # Wykonuje FirstClass.setdata(y,
3.14159)

```

Ani x, ani y nie mają własnego atrybutu `setdata`, dlatego w celu odnalezienia go Python podąża łączem z instancji do klasy. I to mniej więcej wszystko, co można powiedzieć o dziedziczeniu — odbywa się ono w czasie kwalifikowania atrybutów i obejmuje jedynie wyszukiwanie nazw w połączonych obiektach (poprzez podążanie łączami widocznymi na rysunku 27.1).

W funkcji `setdata` wewnętrz klasy `FirstClass` przekazana wartość jest przypisywana do `self.data`. Wewnątrz metody argument `self` — nazwa zgodnie z konwencją nadawana pierwszemu argumentowi wywołania — automatycznie odnosi się do przetwarzanej instancji (x lub y), dzięki czemu przypisania przechowują wartości w przestrzeni nazw instancji, a nie klasy (w taki sposób utworzone zostały zmienne `data` z rysunku 27.1).

Ponieważ klasy mogą generować wiele instancji, metody muszą przetwarzać argument `self` w celu otrzymania przekazanej instancji. Kiedy wywołujemy metodę klasy `display` w celu wyświetlenia `self.data`, widzimy, że dla każdej instancji wynik będzie inny. Z drugiej strony, sama nazwa `display` jest taka sama w x oraz y, ponieważ przychodzi ona (jest dziedziczona) z klasy.

```

>>> x.display()                      # self.data różni się w każdej
instancji
Król Artur
>>> y.display()                      # wywołuje metodę
FirstClass.display()
3.14159

```

Warto zauważyć, że w każdej instancji w składowej `data` przechowaliśmy różne typy obiektów (łańcuch znaków oraz liczbę zmiennoprzecinkową). Tak jak w każdym innym przypadku w Pythonie, deklaracje atrybutów instancji (czasami nazywanych *składowymi*, ang. *member*) nie istnieją. Atrybuty pojawiają się, gdy za pierwszym razem przypiszemy do nich wartości,

podobnie jak proste zmienne. Tak naprawdę gdybyśmy wywołali metodę `display` na jednej z instancji przed wywołaniem metody `setdata`, wywołalibyśmy błąd niezdefiniowanej zmiennej. Atrybut `data` nie istnieje nawet w pamięci, dopóki nie zostanie on przypisany wewnątrz metody `setdata`.

By przekonać się, jak bardzo dynamiczny jest ten model, zobaczymy, że możemy zmodyfikować atrybuty instancji w samej klasie, przypisując wartość do `self` w metodach, lub poza klasą, przypisując ją do jawnego obiektu instancji.

```
>>> x.data = "Nowa wartość"                      # Pobiera lub ustawia atrybuty  
>>> x.display()                                  # Również poza klasą  
Nowa wartość
```

Choć jest to mniej popularne, możemy nawet wygenerować nowy atrybut w przestrzeni nazw, przypisując wartość do jego nazwy poza funkcjami metod klasy.

```
>>> x.anothername = "mielonka"                  # Można tutaj również ustawiać  
nowe atrybuty!
```

Powyższy kod powoduje dołączenie do obiektu instancji `x` nowego atrybutu o nazwie `anothername`, który mógł być wykorzystywany przez któryś z metod klasy, ale nie musiał. Klasy zazwyczaj tworzą wszystkie atrybuty instancji, przypisując do argumentu `self`, jednak nie musi tak być. Programy mogą pobierać, modyfikować lub tworzyć atrybuty dowolnych obiektów, do których mają referencje.

Zazwyczaj nie ma sensu dodawanie danych, których klasa nie może wykorzystać, i można temu zapobiec za pomocą dodatkowego kodu „prywatności” opartego na przeciążeniu operatora dostępu do atrybutów, co omówimy w dalszej części tej książki (patrz rozdział 30. i rozdział 39.). Nie zmienia to jednak faktu, że taki wolny dostęp do atrybutów przekłada się na prostszą składnię, a są przypadki, w których jest nawet użyteczny — na przykład przy kodowaniu rekordów danych, które zobaczymy w dalszej części tego rozdziału.

Klasy dostosowujemy do własnych potrzeb przez dziedziczenie

Poza służeniem jako fabryki generujące wiele obiektów instancji klasy pozwalają nam na modyfikacje za pomocą wprowadzania nowych komponentów (o nazwie *klas podrzędnych*) w miejsce modyfikowania istniejących komponentów w miejscu.

Jak możesz się przekonać, obiekty instancji wygenerowane z klasy dziedziczą atrybuty klasy. Python pozwala również klasom na dziedziczenie po innych klasach, umożliwiając tworzenie *hierarchii* klas specjalizujących jakieś zachowanie za pomocą redefiniowania atrybutów w klasach podrzędnych występujących niżej w hierarchii i przesłaniając tym samym bardziej ogólne definicje tych atrybutów znajdujące się wyżej w drzewie klas. W rezultacie im niżej w hierarchii schodzimy, tym bardziej wyspecjalizowany staje się program. I także tutaj nie można tego porównać z modułami — ich atrybuty istnieją w jednej, płaskiej przestrzeni nazw, która nie jest tak podatna na dostosowywanie do własnych potrzeb.

W Pythonie instancje dziedziczą po klasach, natomiast klasy dziedziczą po klasach nadrzędnych. Poniżej opisano najważniejsze koncepcje leżące u podstaw maszynierii dziedziczenia atrybutów.

- **Klasy nadrzędne są wymieniane w nawiasach w nagłówku instrukcji `class`.** Aby odziedziczyć atrybuty po innej klasie, wystarczy wymienić tę klasę w nawiasach w

nagłówku instrukcji `class`. Klasa dziedzicząca zazwyczaj nazywana jest *klasą podczną* (podklassą), natomiast klasa, z której się dziedziczy — *klasą nadczną* (nadklassą).

- **Klasy dziedziczą atrybuty po swoich klasach nadczędnych.** Podobnie jak instancje dziedziczą nazwy atrybutów zdefiniowane w klasach, tak samo klasy dziedziczą wszystkie nazwy atrybutów zdefiniowane w swoich klasach nadczędnych. Python odnajduje je automatycznie, kiedy próbuje się uzyskać do nich dostęp, o ile nie istnieją one w klasach podczędnych.
- **Instancje dziedziczą atrybuty po wszystkich dostępnych klasach.** Każda instancja otrzymuje zmienne z klasy, z której została wygenerowana, a także ze wszystkich klas nadczędnych tej klasy. Przy szukaniu jakiejś zmiennej Python sprawdzainstancję, później klasę, a następnie wszystkie klasę nadczędne.
- **Każda referencja** obiekt. `atribut` **wywołuje nowe, niezależne wyszukiwanie.** Python wykonuje niezależne wyszukiwanie w drzewie klas dla każdego wyrażenia pobierającego atrybut. Obejmuje to referencje do instancji i klas wykonane poza instrukcjami `class` (na przykład `X.attribut`), a także referencje do atrybutów argumentu instancji `self` w funkcjach metod klasy. Każde wyrażenie `self.attribut` w metodzie wywołuje nowe wyszukiwanie tego atrybutu w `self` oraz wyżej.
- **Zmiany logiki wykonuje się przez tworzenie klas podczędnych, a nie modyfikację klas nadczędnych.** Redefiniując zmienne z klas nadczędnych w klasach znajdujących się niżej w hierarchii (drzewie klas), klasa podczędna zastępuje i tym samym dostosowuje do własnych potrzeb odziedziczone zachowanie.

Rezultat i główny cel wyszukiwania jest taki, że klasy obsługują faktoryzację i dostosowywanie kodu do własnych potrzeb o wiele lepiej niż jakiekolwiek inne narzędzia języka, z jakimi się spotkaliśmy. Z jednej strony, pozwala nam to na minimalizację powtarzalności kodu (i tym samym zredukowanie kosztów jego utrzymania) ze względu na faktoryzację operacji na jedną, współdzieloną implementację. Z drugiej strony, pozwala nam to na programowanie za pomocą dostosowywania do własnych potrzeb czegoś, co już istnieje, zamiast modyfikowania kodu w miejscu lub pisania go od początku.

	Ścisłe mówiąc, <i>dziedziczenie</i> w Pythonie jest nieco bogatsze, niż tutaj zostało opisane, zwłaszcza jeżeli weźmiemy pod uwagę deskryptory i metaklasy w nowym stylu — to bardziej zaawansowane tematy, które omówimy nieco później — ale możemy spokojnie ograniczyć się tutaj tylko do instancji i klas, i to zarówno w tym miejscu naszej książki, jak i w przypadku większości kodu w języku Python. Proces dziedziczenia bardziej formalnie zdefiniujemy w rozdziale 40.
---	---

Drugi przykład

W celu zilustrowania roli dziedziczenia w kolejnym przykładzie rozwijamy kod zaprezentowany wcześniej. Najpierw zdefiniujemy nową klasę `SecondClass`, dziedziczącą wszystkie zmienne klasy `FirstClass` i udostępniającą jedną własną.

```
>>> class SecondClass(FirstClass):          # Dziedziczy metodę setdata
    def display(self):                      # Modyfikuje metodę display
        print('Aktualna wartość = "%s"' % self.data)
```

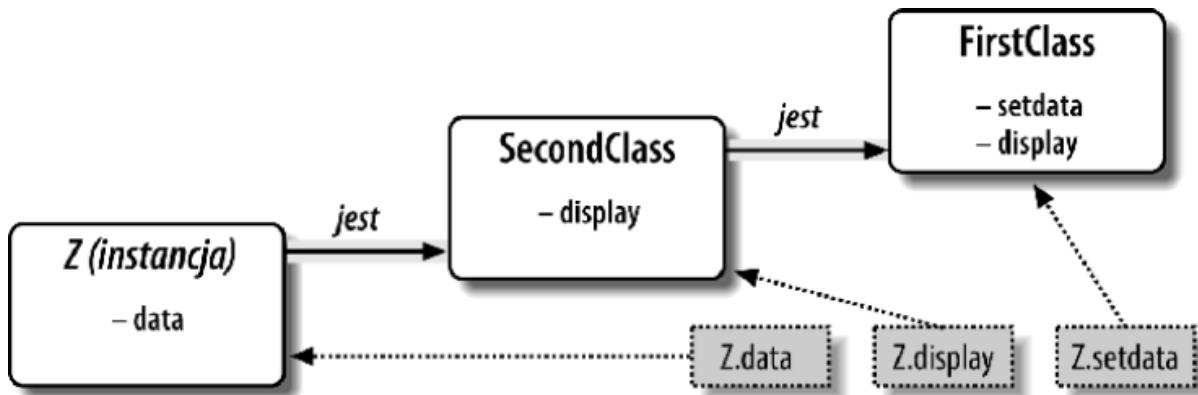
Klasa `SecondClass` definiuje metodę `display` w taki sposób, by wyświetlała ona dane w innym formacie. Definiując atrybut z taką samą nazwą jak atrybut klasy `FirstClass`, `SecondClass` w rezultacie zastępuje atrybut `display` ze swojej klasy nadczędnej.

Warto przypomnieć, że wyszukiwanie dziedziczenia postępuje w górę — od instancji, przez klasy podzielne, do nadzędnych, zatrzymując się na pierwszym wystąpieniu nazwy atrybutu, jaką znajduje. W tym przypadku, ponieważ nazwa `display` z klasy `SecondClass` zostanie odnaleziona przed tą samą nazwą z klasy `FirstClass`, mówimy, że `SecondClass` *przesłania* (zastępuje, nadpisuje, ang. *override*) metodę `display` z `FirstClass`. Czasami takie zastępowanie atrybutów przez ich redefiniowanie niżej nazywamy *przeciążaniem* drzewa.

W rezultacie klasa `SecondClass` specjalizuje `FirstClass`, modyfikując zachowanie metody `display`. Z drugiej strony, `SecondClass` (i wszystkie instancje utworzone z tej klasy) nadal dziedziczy metodę `setdata` w takiej postaci, w jakiej występuje ona w klasie `FirstClass`. Utwórzmy nową instancję, żeby to zademonstrować.

```
>>> z = SecondClass()
>>> z.setdata(42)                                     # Odnajduje metodę setdata w
FirstClass
>>> z.display()                                      # Odnajduje przesłoniętą metodę
w SecondClass
Aktualna wartość = "42"
```

Tak jak wcześniej, tworzymy obiekt instancji `SecondClass`, wywołując tę klasę. Wywołanie metody `setdata` nadal wykonuje wersję z `FirstClass`, jednak tym razem atrybut `display` pochodzi z `SecondClass` i wyświetla zmodyfikowany komunikat. Na rysunku 27.2 przedstawiono przestrzeń nazw, jakie uczestniczą w tych działaniach.



Rysunek 27.2. Specjalizacja: przesłonięcie odziedziczonych nazw przez zredefiniowanie ich w rozszerzeniach niżej w drzewie klas. Na rysunku klasa `SecondClass` redefiniuje i dostosowuje metodę „`display`” do własnych potrzeb jej instancji

A oto bardzo istotna kwestia związana z programowaniem zorientowanym obiektowo. Specjalizacja wprowadzona w klasie `SecondClass` jest całkowicie *zewnętrzna* dla `FirstClass`. Oznacza to, że nie wpływa ona na istniejące lub przyszłe obiekty `FirstClass`, takie jak `x` z poprzedniego przykładu.

```
>>> x.display()                                     # x nadal jest instancją
FirstClass (stary komunikat)
Nowa wartość
```

Zamiast *modyfikować* klasę `FirstClass`, *dostosowujemy* ją do własnych potrzeb. Jest to oczywiście sztuczny przykład, jednak z zasady — ponieważ dziedziczenie pozwala nam na wprowadzanie takich zmian w komponentach zewnętrznych (to znaczy klasach podzielnych) —

klasy często obsługują rozszerzanie i ponowne wykorzystanie kodu o wiele lepiej, niż mogą to robić funkcje oraz moduły.

Klasy są atrybutami w modułach

Zanim przejdziemy dalej, należy pamiętać, że w nazwach klas nie ma nic magicznego. Jest to po prostu zmienna przypisywana do obiektu, kiedy wykonywana jest instrukcja `class`, a do samego obiektu możemy odnosić się za pomocą każdego normalnego wyrażenia. Gdyby na przykład nasza klasa `FirstClass` została utworzona w pliku modułu, a nie wpisana w sesji interaktywnej, moglibyśmy ją zimportować i wykorzystać w normalny sposób w wierszu nagłówka innej klasy.

```
from modulename import FirstClass          # Skopiowanie nazwy do mojego
zakresu

class SecondClass(FirstClass):               # Bezpośrednie wykorzystanie
nazwy klasy

    def display(self): ...
```

Poniższe rozwiązanie dałoby taki sam efekt.

```
import modulename                         # Dostęp do całego modułu

class SecondClass(modulename.FirstClass):   # Składnia kwalifikująca tworzy
referencję

    def display(self): ...
```

Jak wszystko inne, nazwy klas istnieją wewnętrz modułów, dlatego muszą przestrzegać wszystkich reguł omawianych w piątej części książki. W jednym pliku modułu można na przykład zapisać więcej niż jedną klasę — tak jak inne instrukcje modułów, instrukcje `class` wykonywane są w czasie importowania w celu zdefiniowania nazw, a nazwy te stają się osobnymi atrybutami modułu. Mówiąc bardziej ogólnie, każdy moduł może w dowolny sposób mieszać dowolną liczbę zmiennych, funkcji oraz klas, a wszystkie nazwy w module zachowują się w ten sam sposób. Demonstruje to plik `food.py`.

```
# Plik food.py

var = 1                                     # food.var

def func(): ...                               # food.func

class spam: ...                             # food.spam

class ham: ...                               # food.ham

class eggs: ...                            # food.eggs
```

Tak samo będzie nawet wtedy, gdy moduł i klasa noszą tę samą nazwę. Kiedy na przykład mamy poniższy plik `person.py`:

```
class person: ...
```

by pobrać klasę, jak zawsze musimy przejść przez moduł:

```
import person                           # Zimportowanie modułu

x = person.person()                      # Klasa wewnętrz modułu
```

Choć taka ścieżka może się wydawać zbędna, jest konieczna — `person.person` odnosi się do klasy `person` wewnętrz modułu `person`. Samo `person` powoduje pobranie modułu, a nie klasy, o ile oczywiście wcześniej nie użyjemy instrukcji `from`.

```
from person import person # Pobranie klasy z modułu
x = person() # Wykorzystanie nazwy klasy
```

Tak jak w przypadku każdej innej zmiennej, nigdy nie zobaczymy klas z pliku bez uprzedniego zimportowania i pobrania jej w jakiś sposób z pliku, który ją zawiera. Jeśli wydaje się to mylące, nie należy używać tej samej nazwy dla modułu oraz znajdującej się w nim klasy. Tak naprawdę powszechnie stosowana w Pythonie konwencja zaleca rozpoczęwanie nazw klas od *wielkiej litery*, tak by lepiej się one odróżniały.

```
import person # Mała litera w modułach
x = person.Person() # Wielka litera w klasach
```

Należy również pamiętać, że choć klasa i moduły są przestrzeniami nazw służącymi do dołączania atrybutów, odpowiadają różnym strukturom w kodzie źródłowym. Moduł odzwierciedla cały *plik*, natomiast klasa — *instrukcję* wewnętrz tego pliku.Więcej informacji na temat tych różnic znajdziesz później w tej części książki.

Klasy mogą przechwytywać operatory Pythona

Przyjrzyjmy się teraz trzeciej różnicy między klasami a modułami — *przeciążaniu operatorów*. W uproszczeniu przeciążanie operatorów pozwala obiektom zapisanym za pomocą klas przechwytywać i odpowiadać na operacje, które działają na typach wbudowanych — takich jak dodawanie, tworzenie wycinków, wyświetlanie czy kwalifikacja. W dużej mierze jest to mechanizm automatyczny — wyrażenia oraz inne operacje wbudowane kierują sterowanie do implementacji w klasach. I w tym przypadku nie ma czegoś podobnego w modułach — moduły mogą implementować wywołania funkcji, ale nie zachowanie wyrażeń.

Choć moglibyśmy zaimplementować zachowanie klas jako funkcje metod, przeciążanie operatorów pozwala obiektom na ściszęszą integrację z modelem obiektów Pythona. Co więcej, ponieważ przeciążanie operatorów sprawia, że nasze własne obiekty zachowują się jak obiekty wbudowane, powoduje to powstawanie bardziej spójnych i łatwiejszych do zrozumienia interfejsów. Pozwala także na przetwarzanie obiektów opartych na klasach za pomocą kodu napisanego pod kątem interfejsów typów wbudowanych. Poniżej znajduje się krótkie streszczenie najważniejszych koncepcji związanych z przeciążaniem operatorów.

- **Metody zawierające w nazwie podwójne znaki _ (jak `__X__`) są specjalnymi punktami zaczepienia.** Przeciążanie operatorów jest w Pythonie zaimplementowane za pomocą udostępniania metod o specjalnych nazwach, które przechwytyują operacje. Język Python definiuje stałe i niezmienne odwzorowanie każdej z tych operacji na metodę o specjalnej nazwie.
- **Takie metody wywoływane są automatycznie, kiedy instancje pojawią się w operacjach wbudowanych.** Jeśli na przykład obiekt instancji dziedziczy metodę `__add__`, metoda ta wywoływana jest za każdym razem, gdy obiekt ten pojawi się w wyrażeniu ze znakiem `+`. Wartość zwracana przez metodę staje się wynikiem odpowiadającego jej wyrażenia.
- **Klasy mogą nadpisywać większość operacji na typach wbudowanych.** Istnieją dziesiątki specjalnych nazw metod przeciążania operatorów, które służą do przechwytywania i implementowania prawie każdej operacji dostępnej dla typów

wbudowanych. Obejmuje to wyrażenia, ale również podstawowe operacje, takie jak wyświetlanie czy tworzenie obiektów.

- **Nie istnieją wartości domyślne dla metod przeciążania operatorów i nie są one potrzebne.** Jeżeli klasa nie definiuje lub nie dziedziczy metody przeciążania operatorów, oznacza to po prostu, że odpowiadająca jej operacja nie jest obsługiwana na instancjach klasy. Jeżeli na przykład nie ma metody `_add_`, wyrażenia z operatorem + powodują zgłoszenie błędu.
- **Klasy w nowym stylu mają pewne wartości domyślne, ale nie dla typowych operacji.** W Pythonie 3.x i tak zwanych klasach w „nowym stylu” w wersji 2.x, które zdefiniujemy później, klasa główna o nazwie `object` zapewnia wartości domyślne dla niektórych metod `_X_`, ale nie dla wielu i nie dla najczęściej używanych operacji.
- **Operatory pozwalają klasom na integrację z modelem obiektów Pythona.** Przeciążając operacje na typach, obiekty zdefiniowane przez użytkowników zaimplementowane za pomocą klas mogą się zachowywać tak samo jak obiekty wbudowane, co daje nam spójność, a także zgodność z oczekiwanyymi interfejsami.

Przeciążanie operatorów jest opcjonalne. Wykorzystywane jest przede wszystkim przez osoby tworzące narzędzia przeznaczone dla innych programistów Pythona, a nie przez samych twórców aplikacji. I szczerze mówiąc, nie powinno się próbować go używać tylko dla tego, że wydaje się fajne. Dopóki klasa naprawdę nie potrzebuje naśladować wbudowanych interfejsów, powinniśmy się ograniczyć do prostszych nazwanych metod. Po co aplikacja przeznaczona dla bazy danych pracowników miałaby na przykład obsługiwać rozszerzenia takie, jak * oraz +? Nazwane metody, takie jak `giveRaise` oraz `promote`, zazwyczaj mają większy sens.

Z tego powodu nie będziemy się w tej książce zagłębiać w szczegóły każdej metody przeciążania operatorów dostępnej w Pythonie. Istnieje jednak jedna metoda przeciążania operatorów, która występuje w prawie każdej prawdziwej klasie Pythona — metoda `_init_`, znana jako metoda *konstruktora*, wykorzystywana do inicjalizacji stanu obiektów. Powinniśmy zwrócić na nią szczególną uwagę, ponieważ `_init_` wraz z argumentem `self` okazują się kluczami do zrozumienia większości kodu zorientowanego obiektowo w Pythonie.

Trzeci przykład

Przejdzmy do kolejnego przykładu. Tym razem zdefiniujemy klasę podzieloną dla `SecondClass`, implementującą trzy atrybuty o specjalnych nazwach, które Python wywołuje automatycznie:

- Metoda `_init_` wykonywana jest, kiedy konstruowany jest nowy obiekt instancji (`self` jest nowym obiektem klasy `ThirdClass`)^[1].
- Metoda `_add_` wykonywana jest, kiedy instancja tej klasy pojawi się w wyrażeniu z operatorem +.
- Metoda `_str_` wykonywana jest, kiedy obiekt jest wyświetlany (z technicznego punktu widzenia, kiedy przekształcana jest na łańcuch znaków wyświetlania za pomocą wbudowanej funkcji `str` lub jej wewnętrznego odpowiednika w Pythonie).

Nasza nowa klasa podzielona definiuje również normalnie nazwaną metodę `mul`, modyfikującą obiekt instancji w miejscu. Poniżej widać kod nowej podklasy.

```
>>> class ThirdClass(SecondClass):          # Dziedziczy po SecondClass
    def __init__(self, value):                # Przy "ThirdClass(value)"
        self.data = value
    def __add__(self, other):                  # Przy "self + other"
        return ThirdClass(self.data + other)
    def __str__(self):                      # Przy "print(self)", "str()"
```

```

        return '[ThirdClass: %s]' % self.data

    def mul(self, other):
        self.data *= other                                # Modyfikacja w miejscu: nazwana
metoda

>>> a = ThirdClass('abc')                         # Wywołanie metody __init__
>>> a.display()                                  # Wywołanie dziedziczonej
metody

Aktualna wartość = "abc"

>>> print(a)                                     # __str__: zwraca wyświetlany
łańcuch znaków

[ThirdClass: abc]

>>> b = a + 'xyz'                               # __add__: utworzenie nowej
instancji

>>> b.display()                                 # b ma wszystkie metody klasy
ThirdClass

Aktualna wartość = "abcxyz"

>>> print(b)                                     # __str__ zwraca wyświetlany
łańcuch znaków

[ThirdClass: abcxyz]

>>> a.mul(3)                                    # mul: modyfikuje instancję w
miejscu

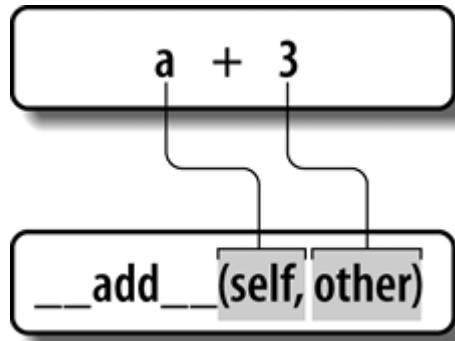
>>> print(a)

[ThirdClass: abcabcabc]

```

Klasa `ThirdClass` jest podklassą `SecondClass`, dlatego jej instancje dziedziczą dostosowaną do własnych potrzeb metodę `display` po tej klasie nadzędnej. Wywołania tworzące obiekty klasy `ThirdClass` przekazują tym razem jednak argument (na przykład `abc`). Jest on przekazywany do argumentu `value` konstruktora `__init__` i przypisywany do atrybutu `self.data`. W rezultacie `ThirdClass` sprawia, że atrybut `data` ustawiany jest automatycznie w czasie tworzenia, nie wymagając już wywoływanego metody `setdata` po tym fakcie.

Co więcej, obiekty klasy `ThirdClass` mogą się teraz pojawiać w wyrażeniach z operatorem `+` oraz wywołaniach funkcji `print`. W przypadku operatora `+` Python przekazuje obiekt instancji znajdujący się po lewej stronie do argumentu `self` w metodzie `__add__`, natomiast wartość z prawej strony do zmiennej `other`, zgodnie z rysunkiem 27.3. Wartość zwracana przez metodę `__add__` staje się wynikiem wyrażenia z operatorem `+` (więcej na temat tego rezultatu już za chwilę).



Rysunek 27.3. W przeciążaniu operatorów operatory wyrażeń oraz inne operacje wbudowane wykonywane na instancjach klasy są odwzorowywane na metody o specjalnych nazwach w klasach. Te metody specjalne są opcjonalne i mogą być dziedziczone w zwykły sposób. Na rysunku wyrażenie ze znakiem + wywołuje metodę `_add_`

W przypadku funkcji `print` Python przekazuje wyświetlany obiekt do `self` w metodzie `_str_`. Łącuch znaków zwracany przez tę metodę przyjmowany jest za wyświetlany łańcuch znaków obiektu. Dzięki metodzie `_str_` (lub bardziej ogólnie bliźniaczej metodzie `_repr_`, którą spotkamy i wykorzystamy w następnym rozdziale) możemy wykorzystać zwykłe wywołanie `print` do wyświetlenia obiektów tej klasy, zamiast wywoływać specjalną metodę `display`.

Metody o specjalnych nazwach, takie jak `_init_`, `_add_` oraz `_str_`, są dziedziczone przez klasy podzielone oraz instancje, podobnie do pozostałych nazw przypisanych w instrukcji `class`. Jeżeli metody nie zostaną zapisane w klasie, Python jak zwykle szuka nazw we wszystkich klasach nadzędnych. Nazwy metod przeciążania operatorów nie są również słowami wbudowanymi lub zastrzeżonymi — są to tylko atrybuty, których Python szuka, kiedy obiekty pojawiają się w różnych kontekstach. Python zazwyczaj wywołuje je automatycznie, jednak czasami mogą one również zostać wywołane przez nasz kod. Na przykład, metoda `_init_` jest na przykład często wywoływana ręcznie w celu uruchomienia konstruktorów klas nadzędnych (więcej informacji na ten temat znajdziesz w kolejnym rozdziale).

Zwracamy wyniki lub nie

Niektóre metody przeciążania operatorów, takie jak `_str_`, wymagają zwracania wyników, ale inne są bardziej elastyczne. Na przykład zwróć uwagę, w jaki sposób metoda `_add_` tworzy i zwraca nowy obiekt instancji swojej klasy, wywołując `ThirdClass` z wartością wynikową — co z kolei uruchamia metodę `_init_` w celu zainicjowania rezultatu. Jest to powszechnie przyjęta konwencja, która wyjaśnia, dlaczego `b` na listingu powyżej posiada metodę `display`; jest to także obiekt klasy `ThirdClass`, ponieważ taki właśnie wynik zwraca + dla obiektów tej klasy.

Dla porównania, metoda `mul` modyfikuje bieżący obiekt instancji w miejscu, przypisując go z powrotem do atrybutu `self`. Moglibyśmy przeciążyć także wyrażenie z operatorem `*`, by robiło to samo, jednak różniłoby się to od działania operatora `*` dla typów wbudowanych, takich jak liczby oraz łańcuchy znaków, dla których zawsze tworzone są nowe obiekty. Przyjęta praktyka zaleca, by przeciążone operatory działały w ten sam sposób, jak działają implementacje operatorów dla typów wbudowanych. Ponieważ przeciążanie operatorów jest tak naprawdę mechanizmem odwzorowania z wyrażenia na metodę, możemy we własnych obiektach klas interpretować operatory w dowolny sposób.

Po co przeciążamy operatory

Jako projektanci klas możemy wybierać, czy chcemy korzystać z przeciążania operatorów, czy też nie. Nasz wybór uzależniony jest od tego, w jak dużym stopniu nasze obiekty mają wyglądać i zachowywać się jak typy wbudowane. Jak wspomniano wcześniej, jeśli pominiemy metodę

przeciążającą operator i nie odziedziczymy jej po klasie nadzędnej, odpowiadającą jej operacja nie będzie obsługiwana w instancjach klasy. Jeśli spróbujemy ją wywołać, otrzymamy wyjątek (lub zostaną użyte standardowe wartości domyślne).

Szczerze mówiąc, wiele metod przeciążania operatorów wykorzystywanych jest tylko wtedy, gdy implementuje się obiekty o matematycznej naturze. Klasa wektora lub macierzy może na przykład przeciążać operator dodawania, natomiast klasa pracownika najprawdopodobniej nie będzie tego robić. W przypadku prostszych klas możemy w ogóle nie korzystać z przeciążania operatorów i zamiast tego polegać na jawnym wywoływaniu metod, które będą implementować zachowanie naszych obiektów.

Z drugiej strony, możemy się zdecydować na wykorzystywanie przeciążania operatorów, jeśli musimy przekazać obiekt zdefiniowany przez użytkownika do funkcji oczekującej zastosowania operatorów dostępnych dla typów wbudowanych, takich jak lista czy słownik. Implementacja tego samego zbioru operatorów w klasie pozwoli zapewnić, że obiekty obsługują te same oczekiwane interfejsy i tym samym są zgodne z funkcją. Choć w niniejszej książce nie omówimy wszystkich metod przeciążania operatorów, kilka dodatkowych technik przeciążania zobaczymy w praktyce w rozdziale 30.

Jedną z często używanych tutaj metod przeciążania jest konstruktor `__init__`, używany do inicjowania nowo utworzonych obiektów instancji i obecny w prawie każdej rzeczywistej klasie. Ponieważ pozwala klasom na natychmiastowe wypełnienie atrybutów w nowo utworzonych instancjach, konstruktor przydaje się w prawie każdym rodzaju klasy, jaki możemy utworzyć. Tak naprawdę nawet jeśli w Pythonie nie deklaruje się atrybutów instancji, zazwyczaj możemy się dowiedzieć, jakie atrybuty będzie miała instancja, sprawdzając kod metody `__init__` jej klasy.

Oczywiście nie ma nic złego w eksperymentowaniu z ciekawymi narzędziami języka programowania, ale nie zawsze przekładają się one na użyteczny kod produkcyjny. W miarę upływu czasu i nabywania doświadczenia przekonasz się, że przedstawione wzorce i praktyki programowania są naturalne i prawie automatyczne.

Najprostsza klasa Pythona na świecie

W tym rozdziale zaczęliśmy szczegółowo omawiać składnię instrukcji `class`, jednak raz jeszcze chciałbym przypomnieć, że podstawowy model dziedziczenia tworzony przez klasy jest bardzo prosty. Tak naprawdę obejmuje on tylko wyszukiwanie atrybutów w drzewach połączonych obiektów. Możemy na przykład utworzyć klasę, w której nie ma niczego. Poniższa instrukcja tworzy klasę bez dołączonych atrybutów (pusty obiekt przestrzeni nazw).

```
>>> class rec: pass # Pusty obiekt przestrzeni nazw
```

Potrzebna nam jest tutaj omówiona w rozdziale 13. instrukcja `pass`, ponieważ nie mamy żadnych metod do zapisania w kodzie. Po utworzeniu klasy za pomocą wpisania tej instrukcji do sesji interaktywnej możemy zacząć dołączać atrybuty do klasy, przypisując do niej zmienne całkowicie poza oryginalną instrukcją `class`.

```
>>> rec.name = 'Edward' # Obiekty z atrybutami  
>>> rec.age = 40
```

Po utworzeniu tych atrybutów przez przypisanie możemy je pobrać za pomocą normalnej składni. Kiedy wykorzystujemy klasę w taki sposób, przypomina ona strukturę z języka C lub rekord z Pascalą. Jest to tak naprawdę obiekt z dołączonymi nazwami pól (podobną pracę możemy wykonać z kluczami słowników, jednak wymaga to dodatkowych znaków).

```
>>> print(rec.name)                                # Jak struktura z języka C lub
rekord
Edward
```

Warto zauważyć, że kod ten działa, nawet jeśli nie istnieją jeszcze żadne instancje klasy. Klasy same w sobie są obiektami, nawet bez instancji. Tak naprawdę są one po prostu samodzielnymi przestrzeniami nazw, dlatego dopóki mamy referencję do klasy, możemy ustawać lub modyfikować jej atrybuty w dowolnym momencie. Zobaczmy jednak, co się dzieje, kiedy utworzymy dwie instancje.

```
>>> x = rec()                                     # Instancje dziedziczą nazwy
klas
>>> y = rec()
```

Te instancje rozpoczynają swoje istnienie jako całkowicie puste obiekty przestrzeni nazw. Ponieważ jednak pamiętają klasę, z jakiej zostały utworzone, otrzymają dołączone do klasy atrybuty za pomocą dziedziczenia.

```
>>> x.name, y.name                            # Zmienna jest przechowywana
tylko w klasie
('Edward', 'Edward')
```

Tak naprawdę instancje nie mają własnych atrybutów — pobierają po prostu atrybut name z obiektu klasy, w którym jest on przechowywany. Jeśli jednak przypiszemy atrybut do instancji, tworzy on (lub modyfikuje) atrybut w tym obiekcie i żadnym innym. Referencje do atrybutów uruchamiają wyszukiwanie dziedziczenia, ale przypisania atrybutów wpływają jedynie na obiekty, w których wykonywane są przypisania. W poniższym kodzie obiekt instancji x otrzymuje własną wartość atrybutu name, natomiast y nadal dziedziczy atrybut name dołączony do klasy znajdującej się nad nim.

```
>>> x.name = 'Amadeusz'                         # Przypisanie modyfikuje jedynie
x
>>> rec.name, x.name, y.name
('Edward', 'Amadeusz', 'Edward')
```

Tak naprawdę, jak zobaczymy w rozdziale 29., atrybuty obiektu przestrzeni nazw są zazwyczaj implementowane jako słowniki, a drzewa dziedziczenia klas są (mówiąc ogólnie) po prostu słownikami z łączami do innych słowników. Jeśli wiemy, gdzie szukać, możemy to zobaczyć.

Na przykład, atrybut `__dict__` jest słownikiem przestrzeni nazw dla większości obiektów opartych na klasach. Niektóre klasy mogą również (lub zamiast tego) definiować atrybuty w słowniku `__slots__`, zaawansowanym i rzadko używanym komponencie, o którym wspomnimy jeszcze w rozdziale 28., ale szczegółowe omówienie odłożymy do rozdziału 31. i rozdziału 32. Zwykle to `__dict__` jest przestrzenią nazw atrybutów instancji.

Aby to zilustrować, w Pythonie 3.3 uruchomiono kod przedstawiony poniżej; kolejność i zbiór wewnętrznych nazw `__X__` może się różnić w zależności od wersji Pythona, dlatego filtryujemy wbudowane nazwy za pomocą wyrażenia generatora, tak jak to zrobiliśmy wcześniej, dzięki czemu wyświetlane są tylko wszystkie przypisane przez nas nazwy.

```
>>> list(rec.__dict__.keys())
['age', '__module__', '__qualname__', '__weakref__', 'name', '__dict__',
 '__doc__']
>>> list(name for name in rec.__dict__ if not name.startswith('__'))
['age', 'name']
```

```

>>> list(x.__dict__.keys())
['name']
>>> list(y.__dict__.keys())                                # list() nie jest wymagane w
Pythonie 2.x
[]

```

W kodzie powyżej w słowniku przestrzeni nazw klasy widoczne są przypisane do niego atrybuty `name` oraz `age`, obiekt `x` ma własny atrybut `name`, a `y` nadal jest pusty. Ze względu na taki model dany atrybut może być często pobierany przez indeksowanie słownika lub notację atrybutu, ale tylko wtedy, gdy jest obecny w danym obiekcie — notacja atrybutu uruchamia wyszukiwanie dziedziczenia, ale indeksowanie przeszukuje *tylko* pojedynczy obiekt (poźniej zobaczysz, że oba sposoby mają swoje odrębne zastosowania):

```

>>> x.name, x.__dict__['name']                         # Atrybuty wymienione tutaj są
kluczami słownika
('Sue', 'Sue')
>>> x.age                                         # ale pobieranie atrybutów
sprawdza również klasy
40
>>> x.__dict__['age']                               # Indeksowanie słownika nie
wykorzystuje dziedziczenia
KeyError: 'age'

```

Aby ułatwić wyszukiwanie dziedziczenia przy pobieraniu atrybutów, każda instancja ma łącze do swojej klasy, które tworzy dla nas Python — nosi ono nazwę `__class__` (to na wypadek, gdybyś miał ochotę je zbadać).

```

>>> x.__class__                                     # łącze instancji do klasy
<class '__main__.rec'>

```

Klasy mają również atrybut `__bases__`, który jest krotką referencji do ich obiektów klasy nadzędnej — w tym przykładzie jest to tylko klasa główna `object` w Pythonie 3.x, o której opowiem nieco później (w wersji 2.x zamiast tego otrzymamy pustą krotkę):

```

>>> rec.__bases__                                    # łącze instancji do klasy do klasy
nadzędnej, () w wersji 2.x
(<class 'object'>,)

```

Dzięki tym dwóm atrybutom widać, jak drzewa klas są reprezentowane w pamięci przez Pythona. Nie musisz jednak znać tego rodzaju wewnętrznych szczegółów — drzewa klas są implikowane przez uruchamiany kod, a ich przeszukiwanie jest zwykle automatyczne — niemniej jednak posiadanie takiej wiedzy powoduje, że ten model staje się nieco mniej tajemniczy.

Najważniejszą informacją, jaką należy zapamiętać, jest to, że model klas Pythona jest bardzo dynamiczny. Klasy oraz instancje są po prostu obiektami przestrzeni nazw z atrybutami tworzonymi w locie przez przypisanie. Takie przypisywanie zazwyczaj mają w kodzie miejsce wewnętrz ciała instrukcji `class`, jednak mogą się pojawić w dowolnym miejscu, w którym mamy referencję do jednego z obiektów drzewa.

Nawet *metody* — normalnie tworzone za pomocą instrukcji `def` osadzonej w instrukcji `class` — można tworzyć całkowicie niezależnie od obiektu klasy. Poniżej znajduje się na przykład kod definiujący poza klasą prostą funkcję przyjmującą jeden argument.

```
>>> def uppername(obj):
    return obj.name.upper()                      # Nadal potrzebuje argumentu self
                                                    (obj)
```

W tym kodzie nie ma jeszcze nic, co by dotyczyło klas — jest to prosta funkcja, która może być w tej chwili wywoływana jako taka, pod warunkiem że przekażemy jej obiekt `obj` z atrybutem `name`; jej wartość z kolei posiada metodę `upper` — instancje naszej klasy mają oczekiwany interfejs i mogą wykonywać konwersję na wielkie litery:

```
>>> upperName(x)                                # Wywołanie jak prostej funkcji
                                                    'AMADEUSZ'
```

Jeśli jednak przypiszemy tę prostą funkcję do atrybutu klasy, staje się ona jej *metodą*, którą można wywołać za pomocą dowolnej instancji, a także przez samą nazwę klasy, o ile instancję przekażemy ręcznie — jest to technika, którą będziemy szerzej wykorzystywać w kolejnym rozdziale^[2].

```
>>> rec.method = uppername                      # Tym razem jest to już metoda
                                                    klasy!
>>> x.method()                                  # Wykonanie metody w celu
                                                    przetworzenia x
                                                    'AMADEUSZ'
>>> y.method()                                  # To samo, jednak przekazuje y
                                                    do self
                                                    'EDWARD'
>>> rec.method(x)                             # Można wywołać przez instancję
                                                    lub klasę
                                                    'AMADEUSZ'
```

Normalnie klasy są tworzone przez instrukcję `class`, a atrybuty instancji są tworzone przez przypisania do atrybutów `self` w funkcjach metod. Tak naprawdę jednak wcale nie musi tak być — programowanie zorientowane obiektowo w Pythonie polega przede wszystkim na wyszukiwaniu atrybutów w połączonych obiektach przestrzeni nazw.

Jeszcze kilka słów o rekordach: klasy kontra słowniki

Choć proste klasy z poprzedniego podrozdziału służą do zilustrowania podstaw modelu klas, stosowane przez nie techniki można wykorzystać do prawdziwej pracy. W rozdziałach 8. i 9. pokazaliśmy, w jaki sposób można użyć słowników, krotek i list do zapisywania właściwości elementów naszego programu; zestawy takich danych nazywamy ogólnie *rekordami*. Okazuje się jednak, że klasy mogą często lepiej sprawdzać się w tej roli — pakują one informacje w sposób podobny do słowników, jednak mogą także dołączać logikę przetwarzania w postaci metod. Dla zilustrowania, poniżej zamieszczamy przykład rekordów utworzonych za pomocą odpowiednio krotek i słownika, które wykorzystywaliśmy już wcześniej:

```
>>> rec = ('Robert', 40.5, ['dev', 'mgr'])      # Rekord oparty na krotce
>>> print(rec[0])
Robert
>>> rec = {}
>>> rec['name'] = 'Robert'                      # Rekord oparty na słowniku
```

```

>>> rec['age'] = 40.5                                # Lub {...}, dict(n=v), itp.
>>> rec['job'] = ['dev', 'mgr']
>>>
>>> print(rec['name'])
Robert

```

Powyższy kod emuluje narzędzia takie jak rekordy z innych języków programowania. Jak jednak widzieliśmy, istnieje kilka sposobów uzyskania tego samego za pomocą klas. Chyba najprostsze będzie poniższe rozwiązanie, zamieniające klucze na atrybuty:

```

>>> class rec: pass

>>> rec.name = 'Robert'                            # Rekord oparty na klasie
>>> rec.age = 40.5
>>> rec.job = ['dev', 'mgr']
>>>
>>> print(rec.name)
Robert

```

Powyższy kod zawiera znacznie mniej składni niż jego odpowiednik oparty na słowniku. Wykorzystuje pustą instrukcję `class` do wygenerowania pustego obiektu przestrzeni nazw. Po utworzeniu pustej klasy wypełniamy ją, przypisując z czasem atrybuty klasy (tak jak wcześniej).

Takie rozwiązanie działa, jednak nowa instrukcja `class` będzie wymagana dla każdego odrębnego rekordu, jaki będzie nam potrzebny. Być może lepiej będzie zamiast tego generować *instancje* pustej klasy reprezentujące każdy z odrębnych elementów:

```

>>> class rec: pass

>>> pers1 = rec()                                  # Informacje zapisane w
instancjach
>>> pers1.name = 'Robert'
>>> pers1.jobs = ['dev', 'mgr']
>>> pers1.age = 40.5
>>>
>>> pers2 = rec()
>>> pers2.name = 'Sonia'
>>> pers2.jobs = ['dev', 'cto']
>>>
>>> pers1.name, pers2.name
('Robert', 'Sonia')

```

W powyższym przykładzie tworzymy dwa rekordy z tej samej klasy. Instancje rozpoczynają swoje istnienie jako puste, podobnie jak klasy. Następnie wypełniamy rekordy za pomocą

przypisania do atrybutów. Tym razem jednak istnieją dwa odrębne obiekty, stąd dwa odrębne atrybuty `name`. Tak naprawdę instancje tej samej klasy nie muszą nawet mieć tego samego zbioru nazw atrybutów. W tym przykładzie jedna z nich ma unikalny atrybut `age`. Instancje są tak naprawdę odrębnymi przestrzeniami nazw, dlatego każda z nich ma odrębny słownik atrybutów. Choć normalnie są one wypełniane w sposób spójny za pomocą metod klas, instancje są o wiele bardziej elastyczne, niż można by się tego spodziewać.

Wreszcie możemy zamiast tego utworzyć w pełni rozbudowaną klasę implementującą rekord oraz jego przetwarzanie — coś, czego słowniki zorientowane na dane nie obsługują bezpośrednio:

```
>>> class Person:
    def __init__(self, name, jobs, age=None):          # Klasa = Dane +
        self.name = name
        self.jobs = jobs
        self.age = age
    def info(self):
        return (self.name, self.jobs)

>>> rec1 = Person('Robert', ['dev', 'mgr'], 40.5)      # Wywołania
konstruktorów
>>> rec2 = Person('Sonia', ['dev', 'cto'])
>>>
>>> rec1.job, rec2.info()                                # Atrybuty + metody
(['dev', 'mgr'], ('Sonia', ['dev', 'cto']))
```

Powыższe rozwiązanie także tworzy większą liczbę instancji, jednak klasa tym razem nie jest pusta — dodaliśmy *logikę* (metody) inicjalizującą instancje w czasie utworzenia i zbierającą atrybuty w krotkę na żądanie. Konstruktor narzuca tutaj spójność instancji, zawsze ustawiając atrybuty `name`, `job` oraz `age`, nawet jeżeli ten ostatni może zostać pominięty podczas tworzenia obiektu. Metody klasy i atrybuty instancji tworzą razem *pakiet* łączący *dane i logikę*.

Moglibyśmy dalej rozszerzać ten kod, dodając do niego logikę obliczającą pensje czy przetwarzającą zmienne. Możemy wreszcie połączyć tę klasę w większą hierarchię w celu odziedziczenia istniejącego zbioru metod za pomocą automatycznego wyszukiwania atrybutów w klasach czy wręcz przechować instancje klasy w pliku za pomocą serializacji z modułu `pickle` Pythona w celu uczynienia ich trwałymi. Tak naprawdę *zrobimy to* w kolejnym rozdziale, gdzie rozwinieśmy analogię między klasami a rekordami w bardziej realistyczny, działający przykład demonstrujący podstawy klas.

Aby być sprawiedliwym w stosunku do innych narzędzi, trzeba zaznaczyć, że w tej formie powyższe wywołania konstrukcji dwóch klas bardziej przypominają słowniki wywołane równocześnie, ale mimo to nadal wydają się być mniej zagracone i zapewniają dodatkowe metody przetwarzania. W rzeczywistości wywołania konstruktorów klasy bardziej przypominają *nazwane krotki* z rozdziału 9. — co ma sens, biorąc pod uwagę, że nazwane krotki są tak naprawdę klasami z dodatkową logiką do mapowania atrybutów na offsety elementów w krotkach:

```
>>> rec = dict(name='Robert', age=40.5, jobs=['dev', 'mgr'])          #
Słowniki
```

```
>>> rec = {'name': 'Robert', 'age': 40.5, 'jobs': ['dev', 'mgr']}
```

```
>>> rec = Rec('Robert', 40.5, ['dev', 'mgr']) # Nazwane krotki
```

Choć typy takie jak słowniki są elastyczne, klasy pozwalają nam dodawać do obiektów działania, których typy wbudowane oraz proste funkcje nie obsługują w sposób bezpośredni. Możemy przechować funkcje także w słownikach, jednak użycie ich w celu przetworzenia domniemanych instancji nie jest aż tak naturalne, jak ma to miejsce w przypadku klas. Aby się o tym przekonać, przejdziemy do kolejnego rozdziału.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadziliśmy podstawy tworzenia klas w Pythonie. Omówiliśmy składnię instrukcji `class` i zobaczyliśmy, jak wykorzystuje się ją w celu utworzenia drzewa dziedziczenia klas. Widzieliśmy również, jak Python automatycznie wypełnia pierwszy argument w funkcjach metod, jak atrybuty dołączane są do obiektów w drzewie klas za pomocą zwykłego przypisania, a także jak noszące specjalne nazwy metody przeciążania operatorów przechwytyują oraz implementują wbudowane operacje dla naszych instancji (na przykład wyrażenia czy wyświetlanie).

Skoro wiemy już wszystko o mechanizmach tworzenia kodu klas w Pythonie, w kolejnym rozdziale przejdziemy do bardziej rozbudowanego i realistycznego przykładu łączącego większość z tego, czego dowiedzieliśmy się dotychczas na temat programowania zorientowanego obiektowo. Później będziemy kontynuować przyglądarkę się tworzeniu kodu klas, raz jeszcze powracając do modelu klas w Pythonie i uzupełniając pominięte tutaj dla uproszczenia szczegóły. Najpierw jednak przejdźmy do quizu, który pozwoli powtórzyć omówione dotychczas zagadnienia.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób klasy powiązane są z modułami?
2. W jaki sposób tworzone są klasy oraz instancje?
3. Gdzie i w jaki sposób tworzone są atrybuty klas?
4. Gdzie i w jaki sposób tworzone są atrybuty instancji?
5. Co w klasie Pythona oznacza `self`?
6. W jaki sposób w kodzie klasy Pythona zapisuje się przeciążanie operatorów?
7. Kiedy możemy chcieć obsługiwać w klasach przeciążanie operatorów?
8. Która metoda przeciążania operatorów wykorzystywana jest najczęściej?
9. Jaki są dwie kluczowe koncepcje niezbędne do zrozumienia kodu zorientowanego obiektowo napisanego w Pythonie?

Sprawdź swoją wiedzę — odpowiedzi

1. Klasy są zawsze zagnieżdżane w modułach; są one atrybutami obiektu modułu. Klasy i moduły są przestrzeniami nazw, jednak klasy odpowiadają instrukcjom (nie całym plikom) i obsługują koncepcje programowania zorientowanego obiektowo takie, jak wiele instancji, dziedziczenie oraz przeciążanie operatorów. W pewnym sensie moduł przypomina klasę z jedną instancją i bez dziedziczenia, odpowiadającą calementu plikowi kodu.
2. Klasy są tworzone poprzez wykonanie instrukcji `class`. Instancje tworzy się, wywołując klasę tak samo, jakby była ona funkcją.
3. Atrybuty klas tworzy się, przypisując atrybuty do obiektu klasy. Normalnie są one generowane przez przypisania najwyższego poziomu zagnieżdżone wewnątrz instrukcji `class`. Każda nazwa przypisana w bloku instrukcji `class` staje się atrybutem obiektu klasy (z technicznego punktu widzenia zakres instrukcji `class` staje się przestrzenią nazw atrybutów obiektu klasy). Atrybuty klas można jednak również tworzyć, przypisując je do klasy w dowolnym miejscu, w którym istnieje obiekt klasy — to znaczy nawet poza instrukcją `class`.
4. Atrybuty instancji tworzy się, przypisując atrybuty do obiektów instancji. Normalnie tworzone są wewnątrz funkcji metod klasy, wewnątrz instrukcji `class`, poprzez przypisanie atrybutów do argumentu `self` (który zawsze jest domniemaną instancją). Mogą one jednak również być tworzone przez przypisanie w dowolnym miejscu, w którym pojawia się referencja do instancji, nawet poza instrukcją `class`. Normalnie wszystkie atrybuty instancji są inicjalizowane w metodzie konstruktora `__init__`. W ten sposób późniejsze wywołania metod mogą zakładać, że atrybuty już istnieją.
5. `self` to nazwa, zgodnie z przyjętą konwencją, nadawana pierwszemu (znajdującemu się najbardziej na lewo) argumentowi funkcji metody klasy. Python automatycznie wypełnia ten argument obiektem instancji, który jest domniemanym podmiotem wywołania metody. Argument ten nie musi nosić nazwy `self` (choć ta konwencja nazewnictwa jest bardzo silna), znaczenie ma tu jego pozycja. Być może dawni programiści języków C++ czy Java wolą nazywać go `this`, ponieważ w tych językach nazwa ta odzwierciedla tę samą koncepcję. W Pythonie jednak argument ten musi być podany w jawnym sposobie.
6. Przeciążanie operatorów w klasach Pythona zapisuje się za pomocą metod o specjalnych nazwach. Wszystkie one zaczynają się i kończą podwójnymi znakami `_`, tak by ich wygląd był unikalny. Nie są to nazwy wbudowane ani zarezerwowane; Python wykonuje je automatycznie, kiedy instancja pojawia się w odpowiadającej im operacji. Sam Python definiuje odwzorowania z operacji na specjalne nazwy metod.
7. Przeciążanie operatorów przydaje się w implementacji obiektów, które przypominają typy wbudowane (na przykład sekwencji czy obiektów numerycznych, takich jak macierze), a także do naśladowania wbudowanych interfejsów typów oczekiwanych przez fragment kodu. Naśladowanie wbudowanych interfejsów typów pozwala nam przekazywać instancje klas mające jednocześnie informacje o stanie — to znaczy atrybuty pamiętające dane pomiędzy wywołaniami operacji. Nie powinniśmy jednak stosować przeciążania operatorów, kiedy wystarczy prosta nazwana metoda.
8. Najczęściej używana jest metoda konstruktora `__init__`. Prawie każda klasa wykorzystuje ją do ustawienia początkowych wartości atrybutów instancji i wykonania innych zadań startowych.
9. Specjalny argument `self` w metodach oraz metoda konstruktora `__init__` to dwa filary kodu zorientowanego obiektowo w Pythonie; jeżeli je zrozumiesz, powinieneś być w stanie zrozumieć większość kodu zorientowanego obiektowo — oprócz wymienionych elementów to praktycznie w większości pakiety funkcji. Oczywiście wyszukiwanie dziedziczenia również ma znaczenie, ale `self` reprezentuje

automatyczny argument obiektu, a konstruktor `__init__` jest stosowany bardzo szeroko.

[1] Nie należy tego mylić z plikami `__init__.py` w pakietach modułów! Metoda tutaj jest funkcją konstruktora klasy używaną do inicjowania nowo utworzonej instancji, a nie pakietu modułu. Więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 24.

[2] Tak naprawdę jest to jeden z powodów, dla których argument `self` musi w metodach Pythona być zawsze jawnym. Ponieważ metody mogą być tworzone jako proste funkcje niezależne od klasy, muszą w sposób jawnym odwoływać się do argumentu domniemanej instancji. Mogą być wywoływane albo jako funkcje, albo jako metody, a Python nie może w inny sposób zgadnąć lub założyć, że prosta funkcja może się w rezultacie stać metodą klasy. Najważniejszym powodem jawnego podawania argumentu `self` jest jednak to, że dzięki niemu znaczenie nazw staje się bardziej oczywiste. Nazwy, do których nie odnosimy się przez `self`, są zwykłymi zmiennymi, natomiast nazwy, do których referencje odbywają się przez `self`, są w sposób oczywisty atrybutami instancji.

Rozdział 28. Bardziej realistyczny przykład

W szczegóły składni klas zagłębimy się bardziej w kolejnym rozdziale. Zanim to jednak nastąpi, chcielibyśmy zaprezentować bardziej realistyczny przykład działania klas, który będzie o wiele bardziej praktyczny od tego, co widzieliśmy dotychczas. W niniejszym rozdziale zbudujemy zbiór klas wykonujących coś o wiele bardziej konkretnego — zapisujących i przetwarzających informacje o ludziach. Jak zobaczymy, to, co w programowaniu w Pythonie nazywamy *klasami* i *instancjami*, często może pełnić te same role, jakie co w bardziej tradycyjnym rozumieniu pełnią *rekordy* i *programy*.

W niniejszym rozdziale utworzymy kod dwóch klas:

- `Person` — klasy tworzącej i przetwarzającej informacje o osobach,
- `Manager` — dostosowanie klasy `Person` do własnych potrzeb, modyfikujące odziedziczone działanie.

Przy okazji będziemy tworzyli instancje obu klas i sprawdzali ich działanie. Po zakończeniu pokażemy interesujący przykładowy przypadek użycia dla klas — przechowamy je w zorientowanej obiektywnie bazie danych typu *shelve*, tak by uczynić je trwałymi. W ten sposób można wykorzystać ten kod jako szablon dla utworzenia pełnej bazy danych osób, napisanej w całości w Pythonie.

Poza próbą pokazania użyteczności tych rozwiązań naszym celem jest także *edukacja*, gdyż rozdział ten stanowi również prezentację programowania zorientowanego obiektywnie w Pythonie. Często zdarza się, że ludzie rozumieją składnię klas z poprzedniego rozdziału na papierze, ale nie wiedzą, od czego zacząć, gdy muszą sami od podstaw napisać kod nowej klasy. W tym celu podzielimy pracę na etapy, tak by móc opanować podstawy. Klasy będąmy budować stopniowo, aby można było zobaczyć, w jaki sposób ich możliwości łączą się ze sobą w pełne programy.

Pod koniec rozdziału nasze klasy będą nadal stosunkowo skromne, jeśli chodzi o ilość kodu, jednak będą demonstrować *wszystkie* najważniejsze koncepcje modelu programowania zorientowanego obiektywnie w Pythonie. Obok szczegółów składni system klas Pythona sprowadza się w dużej mierze do szukania atrybutu w drzewie obiektów wraz ze specjalnym pierwszym argumentem dla funkcji.

Krok 1. — tworzenie instancji

Tyle na temat fazy projektowania — przejdźmy teraz do implementacji. Nasze pierwsze zadanie będzie polegało na rozpoczęciu tworzenia kodu głównej klasy — `Person`. W oknie ulubionego edytora tekstu należy utworzyć nowy plik dla pisanej klasy. W Pythonie silną konwencją jest rozpoczynanie nazw modułów od małych liter, natomiast nazw klas od wielkich liter. Podobnie jak nazwa argumentów `self` w metodach, nie jest to wymagane przez język, jednak na tyle popularne, że zmiana tej konwencji może być myląca dla osób, które później będą odczytywać nasz kod. W celu dostosowania się do tego zwyczaju nazwiemy nasz nowy plik modułu `person.py`, natomiast klasie nadamy nazwę `Person`, jak poniżej:

```
# Plik person.py (początek)
class Person:                                # Utworzenie klasy
```

Aż do pewnego momentu w dalszej części rozdziału wszystkie działania będziemy wykonywać w tym właśnie pliku. W jednym pliku modułu w Pythonie możemy zapisywać kod dowolnej liczby funkcji i klas, jednak nazwa *person.py* może stracić sens, jeśli później będziemy do tego pliku dodawać niezwiązane z nim komponenty. Na razie jednak zakładamy, że cały umieszczony w nim kod będzie powiązany z klasą Person. Tak zresztą powinno być — jak już wiemy, moduły działają najlepiej, jeśli mają jeden, *spójny* cel.

Tworzenie konstruktorów

Pierwszą czynnością, jaką chcemy wykonywać za pomocą klasy Person, jest zapisywanie podstawowych informacji o osobach — na przykład w celu wypełnienia pól rekordu. Oczywiście w Pythonie informacje te znane są pod nazwą *atrybutów* obiektów instancji i tworzone są zazwyczaj za pomocą przypisania do atrybutów *self* w funkcjach metod klas. Normalnym sposobem nadania atrybutom instancji ich pierwszych wartości jest przypisanie ich do *self* w *metodzie konstruktora* *__init__*, która zawiera kod wykonywany przez Pythona — automatycznie za każdym razem, gdy tworzona jest instancja. Dodajmy taką metodę do naszej klasy:

```
# Dodanie inicjalizacji pola rekordu
class Person:
    def __init__(self, name, job, pay):          # Konstruktor przyjmuje 3
                                                # argumenty
        self.name = name                         # Wypełnienie pól przy
                                                    # tworzeniu
        self.job = job                           # self to obiekt nowej
                                                    # instancji
        self.pay = pay
```

Jest to bardzo popularny wzorzec kodu — przekazujemy dane, które będą dołączone do instancji jako argumenty, do metody konstruktora, i przypisujemy je do *self* w celu trwałego ich zachowania. W terminologii programowania zorientowanego obiektowo *self* jest nowo utworzonym obiektem instancji, natomiast *name* (imię i nazwisko), *job* (stanowisko) oraz *pay* (płaca) stają się *informacjami o stanie* — opisowymi danymi zapisanymi w obiekcie w celu późniejszego wykorzystania. Choć inne techniki (takie jak referencje z zakresów zawierających) także są w stanie zapisywać szczegóły, atrybuty instancji sprawiają, że odbywa się to w sposób jawny i łatwy do zrozumienia.

Warto zwrócić uwagę na to, że nazwy argumentów pojawiają się w kodzie *dwukrotnie*. Kod ten na pierwszy rzut oka wydaje się powtarzać, jednak w rzeczywistości tak nie jest. Argument *job* jest na przykład zmienną lokalną w zakresie funkcji *__init__*, natomiast *self.job* jest atrybutem instancji będącej sugerowanym podmiotem wywołania metody. Są to dwie różne zmienne, które — tak się składa — noszą tę samą nazwę. Przypisując zmienną lokalną *job* do atrybutu *self.job* za pomocą kodu *self.job = job*, zapisujemy przekazaną *job* w instancji w celu późniejszego wykorzystania. Jak zawsze w Pythonie miejsce przypisania zmiennej (lub obiekt, do którego jest ona przypisana) determinuje jej znaczenie.

A skoro już mowa o argumentach, w metodzie *__init__* nie ma nic magicznego poza faktem, że jest ona wywoływana automatycznie, kiedy tworzona jest instancja, i ma specjalny pierwszy argument. Pomimo dziwnej nazwy jest to normalna funkcja, która obsługuje wszystkie omówione przez nas możliwości funkcji. Możemy na przykład podać *wartości domyślne* dla

niektórych z jej argumentów, tak by nie trzeba ich było podawać w sytuacjach, gdy ich wartości nie są dostępne bądź przydatne.

W celu zademonstrowania tej możliwości uczynimy argument `job` opcjonalnym. Jego wartością domyślną będzie `None`, co oznacza, że tworzona osoba nie jest (aktualnie) zatrudniona. Jeśli `job` ma wartość domyślną `None`, prawdopodobnie dla porządku będziemy chcieli, by `pay` było równe `0` (o ile oczywiście nie znamy kogoś, kto otrzymuje płacę, nie pracując!). Tak naprawdę musimy dla `pay` podać wartość domyślną, ponieważ zgodnie z regułami składni Pythona i tym, co opisywaliśmy w rozdziale 18., wszystkie argumenty w nagłówku funkcji po pierwszej wartości domyślnej muszą także mieć wartości domyślne.

```
# Dodanie wartości domyślnych do argumentów konstruktora

class Person:

    def __init__(self, name, job=None, pay=0):      # Normalne argumenty
        funkcji

        self.name = name
        self.job = job
        self.pay = pay
```

Kod ten oznacza, że przy tworzeniu obiektów `Person` musimy przekazać imiona i nazwiska (`name`), jednak stanowisko (`job`) oraz płaca (`pay`) są teraz opcjonalne — jeśli je pominiemy, będą miały wartości domyślne, odpowiednio, `None` oraz `0`. Argument `self` jest jak zwykle wypełniany przez Pythona automatycznie, tak by odwoływał się do obiektu instancji. Przypisanie wartości do atrybutów `self` powoduje dołączenie ich do nowej instancji.

Testowanie w miarę pracy

Powyższa klasa niewiele jeszcze robi — na razie wypełnia tylko pola nowego rekordu — jednak jest to prawdziwa, działająca klasa. W tym momencie możemy dodać do niej więcej kodu, by rozszerzyć nieco jej możliwości, jednak nie będziemy tego jeszcze robić. Co każdy już zapewne nauczył się doceniać, programowanie w Pythonie jest tak naprawdę *inkrementalnym prototypowaniem* — piszemy jakiś kod, testujemy go, piszemy więcej kodu, znowu testujemy... Ponieważ Python udostępnia sesję interaktywną i praktycznie natychmiastową reakcję po modyfikacji kodu, testowanie w miarę postępu prac jest o wiele bardziej naturalne od opisania ogromnej ilości kodu w celu wykonania wszystkich testów naraz.

Zanim zatem dodamy do kodu kolejne opcje, przetestujmy to, co już mamy, tworząc kilka instancji naszej klasy i wyświetlając ich atrybuty utworzone przez konstruktor. Moglibyśmy to robić interaktywnie, ale, co już wiadomo na tym etapie książki, testowanie interaktywne ma swoje ograniczenia — ponowne importowanie modułów i wpisywanie przypadków testowych z każdym rozpoczęciem sesji testowania na nowo jest dość żmudne. Częściej programiści Pythona wykorzystują sesję interaktywną w przypadku jednorazowych testów, jednak większe testy wykonują, wpisując kod na dole pliku zawierającego obiekty do testowania, jak poniżej:

```
# Dodanie inkrementalnego kodu autotestu

class Person:

    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
```

```

bob = Person('Robert Zielony')           # Test klasy
anna = Person('Anna Czerwona', job='programista', pay=100000) # Automatycznie
wykonuje __init__
print(bob.name, bob.pay)                  # Pobranie dołączonych atrybutów
print(anna.name, anna.pay)                # Atrybuty dla obiektów „bob” i „anna”
różnią się

```

Warto zauważyć, że obiekt bob przyjmuje wartości domyślne dla atrybutów job oraz pay, natomiast obiekt anna udostępnia własne wartości. Należy także zwrócić uwagę na użycie argumentów ze słowami kluczowymi przy tworzeniu obiektu anna. Moglibyśmy zamiast tego przekazywać wartości przez pozycję, jednak słowa kluczowe pozwolą nam później sobie przypomnieć, czym są dane (a także pozwalają nam przekazywać argumenty w dowolnej kolejności od lewej do prawej strony). I znów, pomimo niezwykłej nazwy, `__init__` jest normalną funkcją, obsługującą wszystko, co już wiemy na temat funkcji — w tym wartości domyślne i przekazywanie argumentów ze słowami kluczowymi.

Kiedy plik ten zostanie wykonany jako skrypt, kod testu znajdujący się na dole tworzy dwie instancje naszej klasy i wyświetla dwa atrybuty każdej z nich (name oraz pay).

```

C:\code> person.py
Robert Zielony 0
Anna Czerwona 100000

```

Kod testu z tego pliku można także wpisać w sesji interaktywnej Pythona (zakładając, że najpierw zimportujemy tam klasę Person), jednak tworzenie gotowych testów wewnątrz modułu, jak powyżej, sprawia, że o wiele łatwiej jest je ponownie wykonać w przyszłości.

Choć kod ten jest stosunkowo prosty, demonstruje już coś istotnego. Jak widać, atrybut name obiektu bob nie jest tym samym co atrybut name obiektu anna; podobnie jest w przypadku atrybutu pay. Każdy z nich jest niezależnym rekordem informacji. Obiekty bob i anna są *obiektemi przestrzeni nazw* — jak wszystkie instancje klasy, każdy z nich ma własną, niezależną kopię informacji o stanie utworzonych za pomocą klasy. Ponieważ każda instancja klasy ma swój własny zbiór atrybutów self, klasy są naturalnym rozwiązaniem służącym do zapisywania w ten sposób informacji dla większej liczby obiektów. Podobnie jak typy wbudowane, klasy służą jako rodzaj *fabryki obiektów*.

Inne struktury programów Pythona, takie jak funkcje i moduły, nie mają takich możliwości. Funkcje domykające z rozdziału 17. są nieco podobne pod względem przechowywania stanu pomiędzy wywołaniami, ale nie posiadają wielu metod, dziedziczenia i rozbudowanej struktury kodu, które otrzymujemy z klas.

Wykorzystywanie kodu na dwa sposoby

W obecnej postaci kod testu znajdujący się na dole pliku działa, jednak jest tu pewien haczyk — instrukcje `print` najwyższego poziomu wykonywane są zarówno wtedy, gdy plik jest wykonywany jako skrypt, jak i gdy importowany jest jako moduł. Oznacza to, że jeśli kiedykolwiek zdecydujemy się zaimportować klasę z tego pliku w celu wykorzystania jej gdzieś indziej (co faktycznie zrobimy w dalszej części rozdziału), będziemy oglądali dane wyjściowe z kodu testu za każdym razem, gdy plik jest importowany. Nie jest to zbyt dobre rozwiązanie w dziedzinie programowania — dla programów klientów nasze wewnętrzne testy nie mają znaczenia i nie chcą one oglądać naszych danych wyjściowych wymieszanych z własnymi.

Choć moglibyśmy wydzielić kod testu do osobnego pliku, często o wiele wygodniejsze jest wpisywanie testów do tego samego pliku co elementy, które są testowane. Lepiej byłoby zorganizować to tak, by wykonywać testy z dołu pliku *tylko* wtedy, gdy plik wykonywany jest w

celach testowych, a nie gdy jest importowany. Właśnie do tego służy sprawdzanie atrybutu `__name__` modułu, co wiemy już z poprzedniej części niniejszej książki. Oto jak będzie wyglądało dodanie go do naszego kodu:

```
# Pozwala na importowanie pliku oraz wykonywanie i testowanie go
class Person:

    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    if __name__ == '__main__':          # Po uruchomieniu w celu przetestowania
        # Kod autotestu klasy
        bob = Person('Robert Zielony')
        anna = Person('Anna Czerwona', job='programista', pay=100000)
        print(bob.name, bob.pay)
        print(anna.name, anna.pay)
```

Teraz otrzymujemy dokładnie takie zachowanie, o jakie nam chodzi. Wykonanie tego pliku w postaci skryptu najwyższego poziomu testuje go, ponieważ jego atrybut `__name__` ma wartość `__main__`, natomiast zimportowanie pliku w postaci biblioteki klas nie wykonuje go.

```
C:\code> person.py
Robert Zielony 0
Anna Czerwona 100000
c:\code> python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) ...
>>> import person
>>>
```

Po zimportowaniu plik definiuje teraz klasę, jednak z niej nie korzysta. Po bezpośrednim wykonaniu plik tworzy dwie instancje naszej klasy, jak wcześniej, i znów wyświetla po dwa atrybuty każdej z instancji, a ponieważ każda instancja jest niezależnym obiektem przestrzeni nazw, wartości ich atrybutów różnią się od siebie.

Zgodność wersji: wyświetlanie

Cały kod zaprezentowany w tym rozdziale działa zarówno w Pythonie 2.x, jak i 3.x, ale w książce używamy go w Pythonie 3.x, a kilka wyników działania używa wywołań funkcji `print` w wersji 3.x z wieloma argumentami. Jak wyjaśniono w rozdziale 11., oznacza to, że niektóre z wyników działania mogą się nieznacznie różnić w Pythonie 2.x. Jeżeli uruchomisz omawiane przykłady w wersji 2.x, sam kod będzie działał w identyczny sposób, ale wokół niektórych wierszy wyników zobaczysz nawiasy, ponieważ dodatkowe nawiasy w funkcji `print` zamieniają kilka elementów w krotkę tylko w wersji 2.x:

```
c:\code> c:\python27\python person.py
('Robert Zielony', 0)
('Anna Czerwona', 100000)
```

Jeżeli takie różnice będą powodować u Ciebie bezsenne noce, po prostu usuń nawiasy, aby użyć instrukcji `print` z wersji 2.x, lub zimportuj funkcję `print` z Pythona 3.x na początku skryptu, jak pokazano w rozdziale 11. (w zasadzie moglibyśmy dodawać takie polecenie we wszystkich przykładach, ale wprowadzałoby to niepotrzebne zamieszanie):

```
from __future__ import print_function
```

Możesz także uniknąć wyświetlania dodatkowych nawiasów w przenoszonym kodzie, używając formatowania do utworzenia wyników w postaci pojedynczego obiektu do wyświetlenia. Poniższe przykłady działają identycznie zarówno w wersji 2.x, jak i 3.x, chociaż wersja z użyciem wywołania metody jest nowszym rozwiązaniem:

```
print('{0} {1}'.format(bob.name, bob.pay))      # Nowa metoda formatująca  
print('%s %s' % (bob.name, bob.pay))           # Wyrażenie formatujące
```

Jak również opisywaliśmy w rozdziale 11., w niektórych przypadkach taki sposób formatowania może być wręcz wymagany, ponieważ obiekty zagnieżdżone w krotce mogą być wyświetlane inaczej niż te wyświetlane jako obiekty najwyższego poziomu — pierwsze z nich są wyświetlane za pomocą wbudowanej funkcji `__repr__`, a te drugie za pomocą funkcji `__str__` (metody przeciążania operatorów zostały omówione w dalszej części tego rozdziału, a także w rozdziale 30.).

Aby uniknąć tego problemu, kody naszych przykładów wyświetlane są za pomocą funkcji `__repr__` (rezerwowa we wszystkich przypadkach, w tym dla zagnieżdżania i sesji interaktywnej) zamiast za pomocą `__str__` (domyślna dla wyświetlania), dzięki czemu zarówno w wersji 3.x, jak i 2.x wszystkie obiekty — nawet takie, które znajdują się w nadmiarowych nawiasach w krotkach — będą wyświetlane tak samo!

Krok 2. — dodawanie metod

Na razie wszystko wygląda świetnie. W tym punkcie nasza klasa jest *fabryką* rekordów, tworzy i wypełnia pola rekordów (w terminologii Pythona — atrybuty instancji). Choć jest dość ograniczona, możemy na jej obiektach wykonywać pewne działania. Klasy dodają dodatkowy poziom struktury, jednak większość swojej pracy wykonują, osadzając i przetwarzając *podstawowe typy danych*, takie jak listy oraz łańcuchy znaków. Innymi słowy, jeśli wiemy już, jak korzystać z podstawowych typów danych Pythona, wiemy także sporo o klasach Pythona — klasy są tak naprawdę niewielkim rozszerzeniem strukturalnym.

Przykładowo pole `name` z naszego obiektu jest prostym łańcuchem znaków, dzięki czemu możemy pobierać nazwiska z naszych obiektów, dzieląc łańcuchy w miejscu wystąpienia spacji i indeksując. Wszystko to są operacje na podstawowych typach danych, które działają bez względzu na to, czy cele ich działania osadzone są w instancjach klas, czy też nie.

```
>>> name = 'Robert Zielony'      # Prosty łańcuch znaków, poza klasą  
>>> name.split()                # Ekstrakcja nazwiska  
['Robert', 'Zielony']  
>>> name.split()[-1]            # Lub [1], jeśli zawsze składa się z 2 części  
'Zielony'
```

W podobny sposób możemy dać danemu obiektowi podwyżkę pensji, aktualizując jego pole `pay` — czyli zmieniając jego informacje o stanie w miejscu za pomocą przypisania. To zadanie obejmuje również proste operacje, które działają na obiektach podstawowych Pythona bez względzu na to, czy są one samodzielne, czy osadzone są w strukturze klasy (formatujemy

poniższy wynik, aby ukryć fakt, że różne wersje Pythona domyślnie wyświetlają różną liczbę cyfr dziesiętnych):

```
>>> pay = 100000          # Prosta zmienna, poza klasą
>>> pay *= 1.10           # 10% podwyżki
>>> print ('%.2f' % pay)    # Lub: pay = pay * 1.10, jeśli lubisz pisać
110000.00                  # Lub: pay = pay + (pay * .10), jeśli
_naprawdę_ lubisz pisać!
```

Aby zastosować takie operacje do obiektów Person utworzonych za pomocą naszego skryptu, wystarczy zrobić z bob.name i anna.pay to samo, co zrobiliśmy przed chwilą z name oraz pay. Operacje te będą te same, jednak obiekty docelowe dołączone są jako atrybuty do obiektów utworzonych na bazie naszej klasy:

```
# Przetworzenie osadzonych typów wbudowanych – łańcuchów znaków; mutowalność
class Person:

    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    if __name__ == '__main__':
        bob = Person('Robert Zielony')
        anna = Person('Anna Czerwona', job='programista', pay=100000)
        print(bob.name, bob.pay)
        print(anna.name, anna.pay)
        print(bob.name.split()[-1])      # Pobranie nazwiska obiektu
        anna.pay *= 1.10                # Danie temu obiektowi podwyżki
        print('%.2f' % anna.pay)
```

Dodaliśmy tutaj trzy ostatnie wiersze. Po wykonaniu pobierają one nazwisko obiektu bob za pomocą prostych działań na łańcuchach znaków i listach, a także dają obiektowi anna podwyżkę, modyfikując jego atrybut pay w miejscu za pomocą prostej operacji na liczbach. W pewnym sensie obiekt anna również jest obiektem *zmiennym* — jego stan zmienia się w miejscu podobnie jak lista po wywołaniu append. A oto wyniki działania nowej wersji:

```
Robert Zielony 0
Anna Czerwona 100000
Zielony
110000.00
```

Powyższy kod działa zgodnie z planem, jednak gdybyśmy pokazali go jakiemuś weteranowi programowania, najprawdopodobniej powiedziałby on nam, że takie ogólne podejście w praktyce nie jest najlepszym pomysłem. Wpiswanie działań takich jak powyższe na stałe, *poza* klasą, może prowadzić do problemów z utrzymaniem kodu w przyszłości.

Co na przykład stanie się, jeśli na stałe zapiszemy wzór kodu pobierającego nazwisko w wielu różnych miejscach programu? Jeśli kiedyś będziemy musieli zmodyfikować jego sposób

działania (na przykład by móc obsługiwać nową strukturę danych osobowych), będziemy musieli odszukać i uaktualnić *każde* jego wystąpienie. W podobny sposób, jeżeli kod obliczający podwykłkę kiedykolwiek się zmieni (na przykład wymagając zgody lub uaktualnienia bazy danych), będziemy musieli zmodyfikować większą liczbę jego kopii. Samo odnalezienie wystąpień takiego kodu może być w większych programach problematyczne — mogą one być rozsiane po wielu plikach czy podzielone na poszczególne kroki. W takich prototypach częste występowanie zmian nie jest niczym nadzwyczajnym.

Tworzenie kodu metod

To, co tak naprawdę chcemy zrobić, to wykorzystać koncepcję z dziedziny programowania znaną pod nazwą *hermetyzowania* (kapsułkowania — ang. *encapsulation*). Hermetyzowanie polega na opakowaniu logiki operacji za interfejsami w taki sposób, by każda operacja była w naszym programie zapisana w kodzie tylko raz. W ten sposób, jeśli nasze potrzeby w przyszłości się zmieniają, trzeba będzie uaktualnić tylko jedną kopię. Co więcej, możemy prawie dowolnie modyfikować zawartość tej pojedynczej kopii bez psucia korzystającego z niej kodu.

W terminologii Pythona chcemy zapisać w kodzie nasze operacje na obiektach w *metodach* klasy, zamiast rozsiewać je po całym programie. Tak naprawdę jest to jedna z rzeczy, do których klasy świetnie się nadają — *faktoryzacja* kodu w celu usunięcia jego powtarzalności i tym samym zoptymalizowania utrzymywania. Dodatkową zaletą jest to, że zmiana działań w metody pozwala na zastosowanie ich do dowolnych instancji klasy, nie tylko do tych, których przetwarzanie zostało zapisane w kodzie na stałe.

Wszystko to w kodzie jest o wiele łatwiejsze, niż brzmi w teorii. Poniżej wykonujemy hermetyzację kodu, przesuwając dwie operacje z kodu poza klasą do metod klasy. A skoro już przy tym jesteśmy, zmodyfikujemy także kod testu samospawdzającego na dole pliku w taki sposób, by wykorzystywał tworzone przez nas nowe metody zamiast zapisanych w kodzie na stałe operacji:

```
# Dodanie metod w celu hermetyzacji operacji i łatwiejszego utrzymania kodu
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):                      # Metody zachowania
        return self.name.split()[-1]           # self to sugerowany
                                             # podmiot
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent)) # Wystarczy zmienić tutaj
    if __name__ == '__main__':
        bob = Person('Robert Zielony')
        anna = Person('Anna Czerwona', job='programista', pay=100000)
        print(bob.name, bob.pay)
        print(anna.name, anna.pay)
        print(bob.lastName(), anna.lastName())      # Użycie nowych metod
```

```

anna.giveRaise(.10)                      # zamiast kodu zapisanego
na stałe

print(anna.pay)

```

Jak już wiemy, *metody* są po prostu normalnymi funkcjami dołączonymi do klas i zaprojektowanymi w taki sposób, by przetwarzać instancje zamiast tych klas. Instancja jest podmiotem wywołania metody i automatycznie przekazywana jest do argumentu `self` metody.

Przekształcenie kodu na metody w tej wersji jest dość łatwe do zrozumienia. Nowa metoda `lastName` robi na przykład z `self` to samo, co poprzednia, zapisana na stałe w kodzie wersja robiła z obiektem `bob`, ponieważ `self` jest sugerowanym podmiotem wywołania metody. Metoda `lastName` zwraca również wynik, ponieważ operacja ta nazywana jest teraz funkcją. Oblicza ona wartość, której może użyć kod wywołujący, nawet jeśli ma ona posłużyć tylko do wyświetlenia. W podobny sposób nowa metoda `giveRaise` robi z `self` to samo, co wcześniej robiliśmy z obiektem `anna`.

Po wykonaniu kodu dane wyjściowe z pliku będą podobne do poprzednich — tak naprawdę w większości dokonaliśmy *refaktoryzacji* kodu w celu ułatwienia zmian w przyszłości, jednak nie modyfikowaliśmy jego sposobu działania:

```

Robert Zielony 0
Anna Czerwona 100000
Zielony Czerwona
110000

```

Warto zwrócić tutaj uwagę na kilka kwestii dotyczących kodu. Po pierwsze, atrybut `pay` obiektu `anna` po podwyżce nadal jest *liczbą całkowitą* — przekształcamy wynik obliczeń z powrotem na liczbę całkowitą, wywołując funkcję wbudowaną `int` wewnętrz metody. Zmiana wartości na `int` (liczbę całkowitą) lub `float` (liczbę zmiennoprzecinkową) w większości sytuacji nie ma dużego znaczenia (obiekty liczb całkowitych i zmiennoprzecinkowych mają te same interfejsy i można je ze sobą mieszać w wyrażeniach), jednak być może w prawdziwym systemie będziemy musieli poradzić sobie z problemem obcinania i zaokrąglania (dla obiektów `Person` pieniądz z pewnością mają znaczenie!).

Jak wiemy z rozdziału 5., możemy sobie z tym poradzić za pomocą funkcji wbudowanej `round(N, 2)`, która pozwoli zaokrąglić i zachować grosze, używając typu `decimal` do określenia precyzji lub przechowując wartości pieniężne w postaci pełnej liczby zmiennoprzecinkowej i wyświetlając je za pomocą łańcucha formatującego `.2f` lub `{0:.2f}` w celu pokazania groszy. W tym przykładzie grosze odetniemy za pomocą funkcji `int`. Aby zobaczyć inne rozwiązanie, zobacz także funkcję `money` w module `formats.py` w rozdziale 25.; możesz zainportować to narzędzie, aby wyświetlać kwoty z przecinkami, groszami i symbolami walut.

Po drugie warto zwrócić uwagę na to, że tym razem wyświetlamy także nazwisko obiektu `anna` — ponieważ logika z nazwiskiem została poddana hermetyzacji w metodzie, możemy ją wykorzystać na *dowolnej instancji* klasy. Jak widzieliśmy, Python mówi metodzie, którą instancję należy przetworzyć, automatycznie przekazując ją do pierwszego argumentu, zazwyczaj noszącego nazwę `self`. A dokładniej:

- W pierwszym wywołaniu — `bob.lastName()` — `bob` jest sugerowanym podmiotem przekazywanym do `self`.
- W drugim wywołaniu — `anna.lastName()` — do `self` trafia zamiast tego `anna`.

Warto prześledzić te wywołania, by zobaczyć, jak instancja trafia do `self` — jest to kluczowa koncepcja. W rezultacie metoda pobiera za każdym razem nazwę sugerowanego podmiotu. To samo dzieje się w przypadku `giveRaise`. Moglibyśmy na przykład dać podwyżkę obiektowi `bob`, wywołując w ten sposób metodę `giveRaise` dla obu instancji. Niestety, początkowa pensja obiektu `bob` w wysokości 0 uniemożliwia mu otrzymanie podwyżki w obecnej wersji programu

(zero pomnożone przez dowolną wartość nadal daje zero; jest to coś, czym możemy się zająć w wersji 2.0 naszego programu).

Wreszcie warto zauważyc, że metoda `giveRaise` zakłada przekazanie zmiennej `percent` w postaci liczby zmiennoprzecinkowej o wartości pomiędzy zero a jedem. W prawdziwym świecie takie założenie może być zbyt radykalne (podwyżka o 1000% zostałaby chyba przez większość z nas uznana za błąd!). Na potrzeby tego prototypu będzie to wystarczające, ale być może będziemy chcieli sprawdzić (a przynajmniej udokumentować) tę kwestię w przyszłej wersji naszego kodu. Powrócimy do tego pomysłu w późniejszym rozdziale książki, w którym zajmiemy się kodem czegoś o nazwie *dekoratory funkcji* i będziemy badać instrukcję `assert`. Oba rozwiązania alternatywne pozwolą nam na automatyczne wykonywanie testów poprawności w trakcie programowania. Na przykład w rozdziale 39. napiszemy narzędzie, które pozwala nam zweryfikować poprawność za pomocą dziwnych inkantacji, takich jak:

```
@rangetest(percent=(0.0, 1.0))          # Używamy dekoratora do sprawdzenia
poprawności
def giveRaise(self, percent):
    self.pay = int(self.pay * (1 + percent))
```

Krok 3. — przeciążanie operatorów

W tym momencie mamy do dyspozycji klasę o pełnych możliwościach, generującą i inicjalizującą instancję, a także dwa nowe elementy działania służące do przetwarzania instancji (w formie metod). Jak na razie wszystko jest w porządku.

W takiej postaci testowanie jest jednak nieco mniej wygodne, niż być powinno — w celu prześledzenia naszych obiektów musimy ręcznie pobierać i wyświetlać *poszczególne atrybuty* (na przykład `bob.name`, `anna.pay`). Byłoby miło, gdyby wyświetlanie instancji w całości od razu dawało nam jakieś przydatne informacje. Niestety, domyślona forma wyświetlania obiektu instancji nie jest zbyt dobra — wyświetla ona nazwę klasy obiektu oraz jego adres w pamięci (co w Pythonie jest właściwie całkowicie zbędne, z wyjątkiem konieczności posiadania unikalnego identyfikatora).

By to zobaczyć, wystarczy zmodyfikować ostatni wiersz w skrypcie na `print(anna)`, tak by wyświetlał on obiekt jako całość. Oto, co otrzymamy — dane wyjściowe mówią, że `anna` jest w wersji 3.x „obiektem”, natomiast w wersji 2.x „instancją”, co zostało pokazane poniżej:

```
Robert Zielony 0
Anna Czerwona 100000
Zielony Czerwona
<__main__.Person object at 0x00000000029A0668>
```

Udostępnienie sposobów wyświetlania

Na szczęście łatwo możemy poprawić tę sytuację, wykorzystując *przeciążanie operatorów* — tworzenie w klasie metod przechwytyjących i przetwarzających wbudowane działania po wykonaniu na instancjach klasy. W tej sytuacji możemy skorzystać z czegoś, co jest chyba najczęściej używaną metodą przeciążania operatorów w Pythonie po metodzie `__init__`: z metodą `__repr__`, którą zaimplementujemy tutaj, oraz jej siostrzanej metody `__str__`, przedstawionej w poprzednim rozdziale.

Wspomniane metody są wykonywane automatycznie za każdym razem, gdy instancja przekształcana jest na ciąg znaków do wyświetlania. Ponieważ to właśnie robi wyświetlenie obiektu za pomocą `print`, w rezultacie wyświetlenie obiektu pokazuje cokolwiek, co zwróci jego metoda `__str__` lub `__repr__`, o ile obiekt ten albo sam ją definiuje, albo dziedziczy po klasie nadrzędnnej (nazwy z podwójnymi znakami `_` dziedziczone są tak samo jak wszystkie pozostałe).

Technicznie rzecz biorąc, metoda `__str__` jest preferowana przez funkcje `print` i `str`, a metoda `__repr__` jest używana jako rezerwowa dla tych ról i we wszystkich innych kontekstach. Mimo że obie mogą być użyte do zaimplementowania różnych sposobów wyświetlania w różnych kontekstach, utworzenie tylko metody `__repr__` wystarczy, aby uzyskać pojedynczy sposób wyświetlania we wszystkich przypadkach — dla funkcji `print`, zagnieżdżeń i echa w sesji interaktywnej. Takie rozwiążanie wciąż pozwala klientom zapewnić alternatywne wyświetlanie za pomocą metody `__str__`, ale tylko w ograniczonych kontekstach; ponieważ w naszym przypadku jest to samodzielny przykład, jest to kwestia dyskusyjna.

Zapisana przez nas już w kodzie metoda konstruktora `__init__` także jest przeciążaniem operatorów — wykonywana jest automatycznie w czasie tworzenia w celu zainicjalizowania nowo utworzonej instancji. Konstruktory są jednak tak często spotykane, że wydają się właściwie przypadkiem specjalnym. Bardziej ograniczone metody, takie jak `__str__`, pozwalają nam wejść w określone operacje i udostępnić *wyspecjalizowane działania*, kiedy nasze obiekty wykorzystywane są w tych kontekstach.

Umieścmy to teraz w kodzie. Poniższy przykład rozszerza naszą klasę w celu uzyskania własnego sposobu wyświetlania, wymieniającego atrybuty, gdy instancje klas wyświetlane są jako całość — w miejsce polegania na mniejszym przydatnym domyślnym sposobie wyświetlania:

```
# Dodanie metody __repr__ przeciążającej operatory w celu wyświetlania
# obiektów

class Person:

    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):                                # Dodana metoda
        return '[Person: %s, %s]' % (self.name, self.pay)  # Wyświetlany
                                                       # ciąg znaków

    if __name__ == '__main__':
        bob = Person('Robert Zielony')
        anna = Person('Anna Czerwona', job='programista', pay=100000)
        print(bob)
        print(anna)
        print(bob.lastName(), anna.lastName())
```

```
anna.giveRaise(.10)  
print(anna)
```

Warto zwrócić uwagę, że w metodzie `__repr__` w celu zbudowania łańcucha wyświetlanego wykorzystujemy łańcuch formatujący %. Klasy wykorzystują wbudowane obiekty typów oraz operacje takie jak ta w celu wykonania swoich zadań. I znów wszystko, czego nauczyliśmy się już o typach wbudowanych oraz funkcjach, ma zastosowanie do kodu opartego na klasach. Klasy w dużej mierze dodają po prostu dodatkową warstwę *struktury*, która opakowuje funkcje i dane razem oraz obsługuje rozszerzenia.

Zmodyfikowaliśmy również nasz kod autotestu, tak by wyświetlał obiekty w sposób bezpośredni, zamiast pokazywać poszczególne atrybuty. Po wykonaniu kodu dane wyjściowe są teraz o wiele bardziej spójne i zrozumiałe. Wiersze ze znakami „[...]” zwracane są przez naszą nową metodę `__repr__`, wykonywaną automatycznie przez operacje wyświetlania.

```
[Person: Robert Zielony, 0]  
[Person: Anna Czerwona, 100000]  
Zielony Czerwona  
[Person: Anna Czerwona, 110000]
```

Wskazówka projektowa: jak dowiesz się w rozdziale 30., metoda `__repr__` jest często używana do niskopoziomowego wyświetlania obiektu jak w kodzie (o ile jest obecna), a metoda `__str__` jest zarezerwowana do wyświetlania danych w sposób bardziej przyjazny dla użytkownika, tak jak to robimy w naszych przykładach. Czasami klasy zapewniają zarówno metodę `__str__` dla wyświetlania danych w sposób przyjazny dla użytkownika, jak i metodę `__repr__`, zapewniającą możliwość wyświetlania dodatkowych szczegółów, co może być przydatne dla programistów. Ponieważ wyświetlanie korzysta z metody `__str__`, a sesja interaktywna wyświetla echo za pomocą metody `__repr__`, może to zapewnić obu docelowym grupom odbiorców odpowiednie do potrzeb sposoby wyświetlania.

Ponieważ metoda `__repr__` dotyczy większej liczby zastosowań, w tym wyświetlania danych zagnieżdżonych, a nie jesteśmy zainteresowani wyświetlaniem dwóch różnych formatów, kompleksowa metoda `__repr__` jest w zupełności wystarczająca dla naszej klasy. Oznacza to również, że nasz niestandardowy sposób wyświetlania będzie używany w wersji 2.x, jeżeli umieścimy obiekty bob i anna w wywołaniu funkcji `print` z wersji 3.x — technicznie są to dane zagnieżdżone, zgodnie z tym, o czym pisaliśmy w ramce „*Zgodność wersji: wyświetlanie*” we wcześniejszej części rozdziału.

Krok 4. — dostosowywanie zachowania za pomocą klas podrzędnych

W tym momencie nasza klasa zawiera większość możliwości programowania zorientowanego obiektowo z Pythona — tworzy instancje, udostępnia działanie w metodach, a nawet wykonuje przeciążanie operatorów w celu przechwytywania operacji wyświetlania za pomocą metody `__repr__`. W rezultacie pakuje nasze dane oraz logikę w jeden samodzielny komponent oprogramowania, ułatwiając zlokalizowanie kodu, a także sprawiając, że jego modyfikacja w przyszłości będzie prosta. Pozwalając na hermetyzację zachowania, pozwala nam także na faktoryzację kodu w celu uniknięcia jego powtarzalności i związanych z tym problemów z utrzymywaniem kodu w przyszłości.

Jedyną ważną koncepcją z dziedziny programowania zorientowanego obiektowo, której nie implementuje jeszcze nasz kod, jest *dostosowywanie do własnych potrzeb za pomocą*

dziedziczenia. W pewnym sensie już korzystamy z dziedziczenia, gdyż instancje dziedziczą metody po klasie. By jednak zaprezentować pełne możliwości programowania zorientowanego obiektowo, musimy zdefiniować związek klasa nadzędna – klasa podrzędna, który pozwoli nam rozszerzać nasze oprogramowanie i zastępować fragmenty odziedziczonego działania. Na tym w końcu polega główna idea programowania zorientowanego obiektowo — dzięki promowaniu modelu opartego na dostosowywaniu do własnych potrzeb już wykonanej pracy pozwala ono drastycznie skrócić czas pisania programów.

Tworzenie klas podrzędnych

W kolejnym kroku zastosujemy nieco metodologii programowania zorientowanego obiektowo w celu wykorzystania naszej klasy Person i dostosowania jej do naszych potrzeb za pomocą rozszerzenia naszej hierarchii oprogramowania. Na potrzeby tego przykładu zdefiniujemy klasę podrzędną Person o nazwie Manager, zastępującą odziedziczoną metodę giveRaise jej bardziej wyspecjalizowaną wersją. Nasza nowa klasa rozpoczyna się w następujący sposób:

```
class Manager(Person):          # Definiuje klasę podrzędną Person
```

Powyższy kod oznacza, że definiujemy nową klasę o nazwie Manager, która dziedziczy po klasie nadzędnej Person i może ją dostosowywać do własnych potrzeb. Upraszczając, klasa Manager jest prawie tym samym co klasa Person, jednak Manager posiada własny sposób przyznawania podwyżek.

Na potrzeby przykładu założymy, że kiedy obiekt klasy Manager otrzymuje podwyżkę, otrzymuje jak zwykle przekazaną wartość procentową, ale także dodatkowy bonus w domyślnej wysokości 10%. Jeśli zatem podwyżka obiektu Manager ustalona została na 10%, tak naprawdę obiekt ten otrzyma 20% (wszelkie podobieństwo do jakichkolwiek osób żyjących lub martwych jest oczywiście przypadkowe). Nasza nowa metoda rozpoczyna się w następujący sposób. Ponieważ ta nowa definicja metody giveRaise będzie w drzewie klas bliższa instancjom klasy Manager niż oryginalna wersja z klasy Person, w rezultacie zastąpi, i tym samym dostosuje do własnych potrzeb, to działanie. Przypomnijmy, że zgodnie z regułami wyszukiwania dziedziczenia, wygrywa wersja nazwy znajdująca się *najniżej* w drzewie:

```
class Manager(Person):          # Dziedziczy atrybuty klasy
Person

    def giveRaise(self, percent, bonus=.10):      # Redefiniuje w celu
dostosowania do własnych potrzeb
```

Rozszerzanie metod — niepoprawny sposób

Istnieją dwa sposoby zapisania tego dostosowania do potrzeb klasy Manager w kodzie — poprawny i niepoprawny. Zaczniemy od *niepoprawnego*, gdyż może on być nieco łatwiejszy do zrozumienia. Niewłaściwy sposób polega na wycięciu i wklejeniu kodu z metody giveRaise klasy Person, a następnie zmodyfikowaniu go w klasie Manager, jak poniżej:

```
class Manager(Person):

    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))      # Źle – wytnij i
wklej
```

Powyższy kod działa zgodnie z planem — kiedy później wywołamy metodę giveRaise instancji klasy Manager, wykonana zostanie jej własna wersja, dorzucająca dodatkowy bonus. Co zatem jest nie tak z kodem, który działa poprawnie?

Problem jest natury ogólnej — za każdym razem, gdy kod kopujemy za pomocą wycinania i wklejania, w rezultacie *podwajamy* wysiłek, jaki będzie w przyszłości niezbędny do jego utrzymywania. Zastanówmy się: ponieważ skopiowaliśmy oryginalną wersję kodu, gdybyśmy kiedyś musieli zmienić sposób przyznawania podwyżek (co raczej na pewno nastąpi), będziemy musieli zmodyfikować kod w dwóch miejscach, a nie tylko w jednym. Choć jest to niewielki i sztuczny przykład, dobrze reprezentuje on problem uniwersalny. Za każdym razem, gdy kusi nas programowanie za pomocą takiego kopowania kodu, najprawdopodobniej powinniśmy poszukać lepszego rozwiązania.

Rozszerzanie metod — poprawny sposób

To, co tak naprawdę chcemy uzyskać, to *rozszerzenie* w jakiś sposób oryginalnej metody `giveRaise` w miejsce całkowitego jej zastąpienia. Poprawnym sposobem wykonania tego w Pythonie jest bezpośrednie wywołanie oryginalnej wersji z rozszerzonymi argumentami, jak poniżej:

```
class Manager(Person):  
    def giveRaise(self, percent, bonus=.10):  
        Person.giveRaise(self, percent + bonus)      # Dobrze – rozszerzenie  
        oryginału
```

Powyższy kod wykorzystuje fakt, że metodę klasy można zawsze wywołać albo za pomocą *instancji* (zwyczajny sposób, gdy Python automatycznie przesyła instancję do argumentu `self`), albo za pomocą *klasy* (mniej popularne rozwiązanie, gdzie instancję trzeba przekazać ręcznie). W bardziej symbolicznej terminologii, przypomnijmy, normalne wywołanie metody w postaci:

`instancja.metoda(argumenty...)`

jest automatycznie tłumaczone przez Pythona na ten odpowiednik:

`klasa.metoda(instancja, argumenty...)`

gdzie klasa zawierająca metodę, która ma być wykonana, ustalana jest za pomocą zastosowania do nazwy metody reguły wyszukiwania dziedziczenia. W kodzie skryptu można zapisać *dowolną* z tych form, jednak jest pomiędzy nimi niewielka różnica — jeżeli chcemy wywoływać metodę bezpośrednio za pomocą klasy, musimy pamiętać o ręcznym przekazaniu instancji. Metoda zawsze potrzebuje podmiotu instancji, przekazanego w ten czy inny sposób, a Python udostępnia go automatycznie jedynie w przypadku wywołań wykonywanych za pośrednictwem instancji. W przypadku wywołań wykonywanych za pomocą nazwy klasy musimy sami przesłać instancję do `self`. W przypadku kodu wewnętrz metody, takiej jak `giveRaise`, `self` jest już podmiotem wywołania i tym samym instancją do przekazania.

Wywołanie bezpośrednio za pomocą klasy w rezultacie odwraca dziedziczenie i wysyła wywołanie w górę drzewa klas w celu wykonania określonej wersji. W naszym przypadku możemy wykorzystać tę technikę do wywołania domyślnej metody `giveRaise` klasy `Person`, nawet jeśli została ona ponownie zdefiniowana na poziomie klasy `Manager`. W pewnym sensie musimy w ten sposób wywoływać metodę za pośrednictwem `Person`, gdyż `self.giveRaise()` wewnętrz kodu metody `giveRaise` klasy `Manager` wykonałby pętlę — ponieważ `self` jest już obiektem `Manager`, `self.giveRaise()` oznaczałoby `Manager.giveRaise` — i tak dalej, rekurencyjnie, aż do wyczerpania dostępnej pamięci.

„Poprawna” wersja może się wydawać niewielką różnicą w kodzie, jednak różnica w zakresie późniejszego *utrzymywania kodu* może być ogromna. Ponieważ logika metody `giveRaise` znajduje się teraz tylko w jednym miejscu (metodzie klasy `Person`), w przyszłości, w miarę ewoluowania potrzeb, będziemy musieli zmodyfikować tylko jedną wersję. I tak naprawdę ta forma w bardziej bezpośredni sposób odpowiada naszym celom — chcemy wykonać

standardowe działanie giveRaise i po prostu dorzucić dodatkowy bonus. Oto nasz pełny plik modułu z zastosowanym nowym krokiem:

```
# Dodanie dostosowania jednego działania do naszych potrzeb w klasie
# podrzędnej

class Person:

    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):

    def giveRaise(self, percent, bonus=.10):          # Redefiniowanie na tym
        poziomie
        Person.giveRaise(self, percent + bonus)         # Wywołanie wersji klasy
    Person

    if __name__ == '__main__':
        bob = Person('Robert Zielony')
        anna = Person('Anna Czerwona', job='programista', pay=100000)
        print(bob)
        print(anna)
        print(bob.lastName(), anna.lastName())
        anna.giveRaise(.10)
        print(anna)

        tom = Manager('Tomasz Czarny', 'manager', 50000) # Utworzenie obiektu
    Manager: __init__
        tom.giveRaise(.10)                                # Wykonanie własnej
    wersji
        print(tom.lastName())                            # Wykonanie
    odziedziczonej metody
        print(tom)                                     # Wykonanie
    odziedziczonej __repr__
```

W celu przetestowania dostosowania klasy podrzędnej Manager do własnych potrzeb dodaliśmy także kod autotestu, tworzący instancję tej klasy, wywołującą jej metody i wyświetlającą dane.

Kiedy tworzymy obiekt klasy Manager, przekazujemy imię i opcjonalnie stanowisko oraz płacę tak jak poprzednio — ponieważ klasa Manager nie ma konstruktora `__init__`, dziedziczy go po klasie Person. Oto wyniki działania nowej wersji kodu:

```
[Person: Robert Zielony, 0]
[Person: Anna Czerwona, 100000]
Zielony Czerwona
[Person: Anna Czerwona, 110000]
Czarny
[Person: Tomasz Czarny, 60000]
```

Wszystko wygląda tutaj świetnie. Obiekty bob i anna są takie same jak poprzednio, a kiedy nowy obiekt klasy Manager, tom, otrzymuje 10% podwyżki, tak naprawdę dostaje 20% (jego pensja zwiększa się z 50 000 do 60 000), ponieważ dostosowana do własnych potrzeb metoda `giveRaise` klasy Manager wykonywana jest tylko dla niego. Warto również zwrócić uwagę, jak wyświetlanie obiektu tom w całości na końcu kodu testu wykorzystuje ładny format zdefiniowany przez metodę `__repr__` klasy Person. Obiekty klasy Manager otrzymują kod tej metody, atrybut `lastName` oraz metodę konstruktora `__init__` „gratis” od klasy Person, dzięki dziedziczeniu.

Polimorfizm w akcji

Aby zaprezentować nabycie odziedziczonego działania w jeszcze bardziej dobitny sposób, możemy na końcu naszego pliku dodać tymczasowo następujący kod:

```
if __name__ == '__main__':
    ...
    print('---Wszystkie trzy---')
    for obj in (bob, anna, tom):          # Ogólne przetwarzanie obiektów
        obj.giveRaise(.10)                 # Wykonanie metody giveRaise tego
                                             # obiektu
    print(obj)                           # Wykonanie wspólnej metody __str__
```

A co z funkcją super?

Aby rozszerzyć dziedziczone metody, przykłady w tym rozdziale po prostu wywołują oryginał za pomocą nazwy klasy nadzędnej: `Person.giveRaise(...)`. Jest to tradycyjny i najprostszy schemat w Pythonie, stosowany w większości przykładów w tej książce.

Programiści Javy mogą być szczególnie zainteresowani faktem, że Python również posiada wbudowaną funkcję `super`, która pozwala na bardziej ogólnie wywoływanie metod klasy nadzędnej — ale korzystanie z niej w wersji 2.x jest dosyć uciążliwe, różni się składnią w wersjach 2.x i 3.x, opiera się na nietypowej semantyce w wersji 3.x, działa niespójnie z przeciążaniem operatorów i nie zawsze dobrze współpracuje z tradycyjnym sposobem kodowania wielokrotnego dziedziczenia, w którym pojedyncze wywołanie klasy nadzędnej nie jest wystarczające.

Na obronę funkcji `super` trzeba jednak powiedzieć, że istnieją również poprawne przypadki jej użycia — dobrze współpracuje z metodami o tej samej nazwie w drzewach wielokrotnego dziedziczenia — ale przy rozwiązywaniu kolejności klas opiera się na mechanizmie MRO (ang. *Method Resolution Order*), który wielu programistów uważa za ezoteryczny i sztuczny; w nierealistyczny sposób zakłada ona, że uniwersalny sposób wdrożenia jest zawsze niezawodny; nie obsługuje w pełni zastępowania metod i

modyfikacji list argumentów i dla wielu użytkowników wydaje się być niezbyt przejrzystym rozwiązaniem, które jest rzadko spotykane w prawdziwym kodzie Pythona. Ze względu na te wady w naszej książce preferujemy jawnie wywoływanie klas nadzędnych za pomocą nazwy zamiast korzystania z funkcji super, zalecamy tę samą zasadę początkującym użytkownikom i odkładamy prezentację funkcji super aż do rozdziału 32. Powinieneś poczekać z oceną tej funkcji aż do chwili, kiedy poznasz prostsze i ogólnie bardziej tradycyjne „pythonowe” sposoby realizacji takich zadań, zwłaszcza gdy dopiero zaczynasz stawiać swoje pierwsze kroki w programowaniu zorientowanym obiektowo. Tematy takie jak mechanizm MRO i wspólne dziedziczenie wielokrotne wydają się być bardzo wymagającymi zagadnieniami dla początkujących programistów i nie tylko.

Na koniec jedna uwaga dla wszystkich programistów Javy zaczynających przygodę z Pythonem: spróbujcie oprzeć się pokusie używania funkcji super Pythona, dopóki nie będziecie mieć okazji dogłębnego przestudiowania jej subtelnego implikacji. Gdy przejdziecie do wielokrotnego dziedziczenia z funkcją super, zwykle nie będzie ona działać tak, jak myślicie, a często będzie robić więcej, niż się spodziewacie. Klasa, która będzie wywoływać, może wcale nie być klasą nadzczną, a nawet może różnić się w zależności od kontekstu wywołania. Parafrując cytat ze słynnego filmu Roberta Zemeckisa: funkcja super w Pythonie jest jak pudełko czekoladek — *nigdy nie wiesz, na co trafisz!*

Oto uzyskany rezultat, z nowymi elementami wyróżnionymi pogrubioną czcionką:

```
[Person: Robert Zielony, 0]  
[Person: Anna Czerwona, 100000]  
Zielony Czerwona  
[Person: Anna Czerwona, 110000]  
Czarny  
[Person: Tomasz Czarny, 60000]  
--Wszystkie trzy--  
[Person: Robert Zielony, 0]  
[Person: Anna Czerwona, 121000]  
[Person: Tomasz Czarny, 72000]
```

W dodanym kodzie obiekt jest *albo* instancją klasy Person, *albo* klasy Manager, a Python automatycznie wykonuje kod odpowiedniej metody giveRaise — oryginalną wersję z Person dla obiektów bob oraz anna, a wersję zmodyfikowaną z Manager dla obiektu tom. Warto samodzielnie prześledzić te wywołania metod w celu zobaczenia, jak Python wybiera właściwą metodę giveRaise dla każdego obiektu.

Jest to po prostu przykład działania zjawiska *polimorfizmu* Pythona, z którym spotkaliśmy się wcześniej — to, co robi giveRaise, uzależnione jest od tego, na czym wykonujemy tę metodę. W powyższym kodzie jest to jeszcze bardziej oczywiste, kiedy metoda ta wybiera z kodu, który zapisaliśmy sami w klasach. Praktycznym efektem tego kodu jest to, iż obiekt anna otrzymuje następne 10% podwyżki, natomiast obiekt tom otrzymuje kolejne 20%, ponieważ metoda giveRaise wybierana jest w oparciu o typ obiektu. Jak już wiemy, polimorfizm jest sercem elastyczności Pythona. Przekazanie dowolnego z trzech naszych obiektów do funkcji wywołującej metodę giveRaise miałoby ten sam efekt — w zależności od typu przekazanego obiektu automatycznie wykonana została właściwa wersja.

Z drugiej strony, wyświetlanie za pomocą print wykonuje *tę samą* metodę `_str_` dla wszystkich trzech obiektów, ponieważ kod tej metody został napisany tylko raz w klasie Person.

Klasa Manager specjalizuje i stosuje kod napisany oryginalnie w Person. Choć przykład ten jest niewielki, wykorzystuje możliwości programowania zorientowanego obiektowo w zakresie dostosowywania kodu do własnych potrzeb i ponownego wykorzystania go. W przypadku klas czasami wydaje się to działać całkowicie automatycznie.

Dziedziczenie, dostosowanie do własnych potrzeb i rozszerzenie

Tak naprawdę klasy mogą być jeszcze bardziej elastyczne, niż sugeruje to nasz przykład. Klasa Person może dziedziczyć, dostosowywać do własnych potrzeb lub rozszerzać istniejący kod z klas nadzędnych. Przykładowo, choć w tekście skupiliśmy się na aspekcie dostosowania do własnych potrzeb, do klasy Manager możemy także dodać unikalne metody, których nie ma w klasie Person, jeśli obiekty tej klasy wymagają czegoś zupełnie z innej beczki (odwołanie do *Latającego Cyrku Monty Pythona* jest zamierzzone). Dobra ilustracja to poniższy fragment kodu. Metoda giveRaise redefiniuje tutaj metodę z klasy nadzędnej w celu dostosowania jej do własnych potrzeb, natomiast somethingElse definiuje coś nowego w celu rozszerzenia klasy:

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __repr__(self): ...

class Manager(Person):                      # Dziedziczenie
    def giveRaise(self, ...): ...             # Dostosowanie do własnych
    potrzeb                                  # Rozszerzenie

    def somethingElse(self, ...): ...         # Rozszerzenie

tom = Manager()
tom.lastName()                            # Odziedziczona wprost
tom.giveRaise()                           # Wersja dostosowana do
własnych potrzeb                         # Tutaj rozszerzenie
tom.somethingElse()                      # Odziedziczona przeciążona
print(tom)                                # metoda
```

Dodatkowe metody, jak somethingElse z powyższego kodu, rozszerzają istniejące oprogramowanie i dostępne są tylko dla obiektów klasy Manager, a nie dla obiektów klasy Person. Na cele tego przykładu ograniczymy nasze czynności do dostosowywania działania klasy Person do naszych potrzeb za pomocą ponownego zdefiniowania ich w klasie podzędnej, a nie dodawania czegoś do niej.

Programowanie zorientowane obiektowo — idea

Nasz kod, choć niewielki, jest całkiem funkcjonalny. I tak naprawdę ilustruje już najważniejszą kwestię związaną ogólnie z programowaniem zorientowanym obiektowo: programujemy za pomocą *dostosowywania* tego, co zostało już wykonane, do własnych potrzeb, a nie za pomocą kopiowania czy modyfikowania istniejącego kodu. Nie jest to zawsze na pierwszy rzut oka oczywiste dla osób początkujących, zwłaszcza biorąc pod uwagę dodatkowe wymagania w

zakresie kodu klas. Styl programowania powiązany z klasami może jednak zdecydowanie skrócić czas programowania w porównaniu z innymi rozwiązaniami.

W naszym przykładzie moglibyśmy teoretycznie zaimplementować własne działanie `giveRaise` bez tworzenia klasy podzielnej, jednak żadna z poniższych opcji nie daje kodu tak optymalnego jak nasz:

- Choć moglibyśmy po prostu napisać kod dla obiektów `Manager` od podstaw, jako nowy, samodzielny kod klasy, musielibyśmy ponownie zaimplementować wszystkie działania klasy `Person`, które dla obiektów `Manager` są takie same.
- Choć moglibyśmy po prostu zmodyfikować istniejącą klasę `Person` w miejscu pod kątem wymagań metody `giveRaise` klasy `Manager`, takie działanie najprawdopodobniej zniszczyłoby miejsca, w których nadal potrzebne jest nam oryginalne działanie klasy `Person`.
- Choć moglibyśmy po prostu skopiować klasę `Person` w całości, zmienić nazwę jej kopii na `Manager` i zmodyfikować jej metodę `giveRaise`, takie działanie wprowadziłoby powtarzalność kodu i w przyszłości podwoiłoby naszą ilość pracy — modyfikacje wprowadzone do klasy `Person` nie zostałyby pobrane automatycznie i musielibyśmy je ręcznie przenieść do kodu klasy `Manager`. Jak zwykle podejście „wytnij i wklej” teraz może się wydawać szybkie, ale podwaja ilość pracy potrzebnej w przyszłości.

Hierarchia kodu, który możemy dostosować do własnych potrzeb, budowana za pomocą klas, jest o wiele lepszym rozaniem w przypadku oprogramowania, które ewoluje z czasem. Żadne inne narzędzia Pythona nie obsługują tego trybu programowania. Ponieważ możemy przycinać i rozszerzać wykonaną już pracę, dodając kod nowych klas podzielnych, możemy skorzystać z tego, co zostało już zrobione, zamiast za każdym razem rozpoczynać od podstaw, niszcząc to, co już działa, lub wprowadzać kilka kopii kodu, które w przyszłości będziemy musieli uaktualnić. Wykonane w poprawny sposób programowanie zorientowane obiektywne jest wielkim sojusznikiem programisty.

Krok 5. — dostosowanie do własnych potrzeb także konstruktorów

Nasz kod w obecnej postaci działa, jednak jeśli przyjrzymy się bliżej jego aktualnej wersji, pewien element może wydać nam się nieco dziwny. Podanie nazwy stanowiska 'manager' dla obiektu `Manager` przy tworzeniu wydaje się bez sensu — nazwa stanowiska wynika przecież z samej klasy. Lepiej byłoby, gdybyśmy mogli w jakiś sposób wypełnić tę wartość automatycznie przy tworzeniu obiektu tej klasy.

Sztuczka, jaką wykorzystamy do poprawienia tego kodu, okazuje się *tą samą*, jaką wykorzystaliśmy w poprzednim podrozdziale. Chcemy dostosować logikę konstruktora klasy `Manager` do własnych potrzeb w taki sposób, by nazwa stanowiska dodawana była automatycznie. Jeśli chodzi o kod, musimy zdefiniować metodę `__init__` klasy `Manager`, tak by przekazywała ona łańcuch znaków 'manager' za nas. I tak jak w przypadku dostosowania do naszych potrzeb metody `giveRaise`, chcemy wykonać oryginalną metodę `__init__` klasy `Person`, wywołując ją za pośrednictwem nazwy klasy, tak by nadal inicjalizowała ona atrybuty informacji o stanie.

Poniższe rozszerzenie kodu `person.py` wykona wyznaczone zadanie. Utworzyliśmy nowy konstruktor klasy `Manager` i zmodyfikowaliśmy wywołanie tworzące obiekt `tom`, tak by nie przekazywało ono nazwy stanowiska 'manager'.

```
#Plik person.py
```

```
# Dodanie dostosowanego do własnych potrzeb konstruktora w klasie podzielnej
```

```

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)
class Manager(Person):
    def __init__(self, name, pay):                      # Zredefiniowanie
        konstruktora
            Person.__init__(self, name, 'manager', pay) # Wykonanie oryginalnej
        metody z łańcuchem
                                            # 'manager'

    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob)
    print(anna)
    print(bob.lastName(), anna.lastName())
    anna.giveRaise(.10)
    print(anna)

    tom = Manager('Tomasz Czarny', 50000)          # Nazwa stanowiska nie jest
    potrzebna:
        tom.giveRaise(.10)                          # jest ona sugerowana
    (ustawiana) za pomocą klasy

    print(tom.lastName())
    print(tom)

```

Do rozszerzenia konstruktora `__init__` wykorzystujemy tutaj tę samą technikę, którą zastosowaliśmy wcześniej w przypadku metody `giveRaise` — wykonanie wersji z klasy nadrzednej za pomocą wywołania za pośrednictwem nazwy klasy i przekazania instancji `self` w sposób jawnny. Choć konstruktor ma dziwną nazwę, rezultat jest identyczny. Ponieważ potrzebowaliśmy wykonać także logikę konstruktora klasy `Person` (w celu zainicjalizowania

atrybutów instancji), naprawdę musielibyśmy wywołać metodę w ten sposób — w przeciwnym razie instancje nie miałyby dołączonych żadnych atrybutów.

Wywołanie konstruktora klasy nadrzędnej z jego zdefiniowanego na nowo odpowiednika w ten sposób okazuje się w Pythonie bardzo popularnym wzorcem kodu. Sam z siebie Python wykorzystuje dziedziczenie do wyszukania i wywołania tylko jednej metody `__init__` naraz w czasie tworzenia instancji — metody znajdującej się *najniżej* w drzewie klas. Jeśli chcemy, by w czasie konstrukcji obiektu wykonane zostały metody `__init__` znajdujące się wyżej w drzewie (a zazwyczaj tak jest), musimy wywołać je ręcznie za pośrednictwem nazwy klasy nadrzędnej. Zaletą tego rozwiązania jest to, że możemy w jawny sposób zdecydować, który argument przekazać do konstruktora klasy nadrzędnej, a także możemy wybrać, by wcale go *nie* wywoływać. Niewywołanie konstruktora klasy nadrzędnej pozwala w całości zastąpić jego logikę, zamiast tylko ją rozszerzyć.

Wynik kodu testu samosprawdzającego pliku będzie taki sam jak ostatnio — nie zmieniliśmy działania kodu, a jedynie poddaliśmy go restrukturyzacji w celu pozbycia się powtarzalności.

```
[Person: Robert Zielony, 0]  
[Person: Anna Czerwona, 100000]  
Zielony Czerwona  
[Person: Anna Czerwona, 110000]  
Czarny  
[Person: Tomasz Czarny, 60000]
```

Programowanie zorientowane obiektowo jest prostsze, niż się wydaje

W swojej obecnej postaci, pomimo rozmiaru, nasze klasy wykorzystują prawie wszystkie najważniejsze koncepcje z dziedziny programowania zorientowanego obiektowo w Pythonie:

- tworzenie instancji — wypełnianie atrybutów instancji,
- metody i działanie — hermetyzacja logiki w metodach klas,
- przeciążanie operatorów — udostępnianie działania operacjom wbudowanym, takim jak wyświetlanie,
- dostosowanie działania do własnych potrzeb — redefiniowanie metod w klasach podrzędnych w celu ich specjalizacji,
- dostosowanie konstruktorów do własnych potrzeb — dodanie logiki inicjalizującej do kroków z klasy nadrzędnej.

Większość z tych koncepcji oparta jest na trzech podstawowych ideach: wyszukiwaniu dziedziczenia atrybutów w drzewie obiektów, specjalnym argumencie `self` w metodach oraz automatycznym udostępnianiu przeciążania operatorów w metodach.

Przy okazji sprawiliśmy także, że nasz kod będzie łatwy do zmodyfikowania w przyszłości, wykorzystując skłonność klas do faktoryzowania kodu w celu zmniejszenia jego *powtarzalności*. Przykładowo opakowaliśmy logikę w metody i wywołujemy metody klas nadrzędnych z ich rozszerzeń w celu uniknięcia posiadania kilku kopii tego samego kodu. Większość z tych kroków była naturalną konsekwencją strukturyzujących możliwości klas.

Ogólnie rzecz biorąc, na tym właśnie polega programowanie zorientowane obiektowo w Pythonie. Klasy z pewnością mogą stać się większe od tych z przykładu. Istnieją również bardziej zaawansowane zagadnienia związane z klasami, takie jak dekoratory i metaklasy, z którymi spotkamy się w późniejszych rozdziałach. Jeśli jednak chodzi o podstawy, nasze klasy robią wszystko, co należy. Tak naprawdę dla każdej osoby, która pojęła sposób działania

napisanych przed chwilą klas, większość kodu z dziedziny programowania zorientowanego obiektowo w Pythonie powinna być teraz na wyciągnięcie ręki.

Inne sposoby łączenia klas

To powiedziawszy, powiniensem również wspomnieć o tym, że choć podstawy programowania zorientowanego obiektowo są w Pythonie proste, w większych programach sztuka polega na tym, w jaki sposób te klasy ze sobą łączymy. W tym przykładzie skupialiśmy się na *dziedziczeniu*, ponieważ mechanizm ten udostępniany jest przez Pythona, jednak programiści czasami łączą klasy również na inne sposoby.

Przykładowo popularny wzorzec programowania obejmuje zagłędzanie obiektów wewnątrz siebie w celu tworzenia tak zwanych *kompozytów* (ang. *composite*). Wzorzec ten omówimy bardziej szczegółowo w rozdziale 30., który poświęcony jest bardziej projektowaniu niż samemu Pythonowi. Moglibyśmy jednak wykorzystać ideę kompozycji w naszym rozszerzeniu Manager, *osadzając* klasę Person, zamiast po niej dziedziczyć.

Poniższe rozwiązanie alternatywne, zapisane w pliku *person-composite.py*, robi to za pomocą wykorzystania metody przeciążania operatorów `__getattr__` do przechwytywania prób pobierania niezdefiniowanych atrybutów i delegowania ich do osadzonego obiektu za pomocą wbudowanej funkcji `getattr`. Wywołanie funkcji `getattr` zostało wprowadzone w rozdziale 25. — jest takie samo jak notacja `X.Y` pobierania atrybutów, a zatem przeprowadza dziedziczenie, z tym że nazwa atrybutu `Y` jest łańcuchem definiowanym podczas działania programu — funkcja `__getattr__` została w pełni opisana w rozdziale 30., ale jej podstawowa składnia jest wystarczająco prosta, aby ją tutaj wykorzystać.

Poprzez połączenie tych narzędzi metoda `giveRaise` nadal służy do dostosowania do własnych potrzeb, modyfikując argument przekazywany do osadzonego obiektu. W rezultacie klasa Manager staje się warstwą kontrolera przekazującą wywołania *w dół*, do osadzonego obiektu, zamiast *w górę*, do metod klasy nadzędnej:

```
# Plik person-composite.py

# Alternatywa klasy Manager z osadzaniem

class Person:
    ...to samo...

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay)      # Osadzenie obiektu
    Person

    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)           # Przechwycenie i
    delegowanie

    def __getattr__(self, attr):
        return getattr(self.person, attr)               # Delegowanie wszystkich
    pozostałych atrybutów

    def __repr__(self):
        return str(self.person)                         # Musi znowu przeciążać
    operator (w 3.x)

    if __name__ == '__main__':
```

...to samo...

Wyniki działania tej wersji są identyczne jak poprzednio, stąd nie będziemy ich tutaj ponownie przedstawiać. O wiele ważniejsze jest jednak to, że powyższa alternatywa klasy Manager jest reprezentatywna dla ogólnego wzorca kodu zazwyczaj znanego pod nazwą *delegacji* — struktury opartej na kompozycie, która zarządza opakowanym obiektem i przesyła do niego wywołania metod.

Wzorzec ten działa w naszym przykładzie, jednak wymaga mniej więcej dwa razy więcej kodu i jest gorzej od dziedziczenia przystosowany do takich typów bezpośredniego dostosowywania do własnych potrzeb, jakie chcieliśmy uzyskać (tak naprawdę żaden rozsądny programista Pythona w praktyce nigdy nie zapisałby tego przykładu w ten sposób, z wyjątkiem, być może, ogólnych celów demonstracyjnych). Obiekt Manager nie jest tutaj tak naprawdę obiektem Person, dlatego potrzebny jest nam dodatkowy kod w celu ręcznego przekazania wywołań metod do osadzonego obiektu. Metody przeciążania operatorów, takie jak `__repr__`, musimy zdefiniować ponownie (przynajmniej w wersji 3.x, co wynika z ramki „Przechwytywanie wbudowanych atrybutów w wersji 3.x” w dalszej części rozdziału), a dodanie nowych działań dla klasy Manager jest o wiele mniej oczywiste, ponieważ informacje o stanie zostały odsunięte o jeden poziom.

Mimo to *osadzanie obiektów* i oparte na nim wzorce projektowe mogą być dobrym rozwiązaniem, kiedy osadzone obiekty wymagają bardziej ograniczonej interakcji z zawierającym je pojemnikiem niż w przypadku bezpośredniego dostosowania do własnych potrzeb. Warstwa kontrolera, taka jak w tej alternatywnej klasie Manager, mogłaby się na przykład przydać, gdybyśmy chcieli zaadaptować klasę do oczekiwanej interfejsu, który nie jest przez nią obsługiwany, śledzić wywołania metod innego obiektu lub sprawdzać ich poprawność. I faktycznie, prawie identyczny wzorzec kodu wykorzystamy, kiedy będziemy omawiać *dekoratory klas* w dalszej części książki.

Co więcej, hipotetyczna klasa `Department` („dział”), podobna do poniższej, mogłaby *agregować* inne obiekty w celu potraktowania ich jako zbiór. Spróbuj samodzielnie zamienić kod autotestu z końca pliku `person.py` na poniższy kod, by to wypróbować (gotowy przykład znajdziesz w pliku `person-department.py` w zestawie przykładów do książki):

```
# Plik person-department.py

# Agregacja osadzonych obiektów w kompozyt

class Person:
    ...to samo...

class Manager(Person):
    ...to samo...

class Department:
    def __init__(self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
```

```

        print(person)

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    sue = Person('Anna Czerwona', job='dev', pay=100000)
    tom = Manager('Tomasz Czarny', 50000)
development = Department(bob, anna)      # Osadzenie obiektów w kompozycie
development.addMember(tom)
development.giveRaises(.10)              # Wykonuje metodę giveRaise osadzonych
                                         # obiektów
development.showAll()                   # Wykonuje metody __repr__ osadzonych
                                         # obiektów

```

Przechwytywanie wbudowanych atrybutów w wersji 3.x

Uwaga dotycząca implementacji: w Pythonie 3.x (a także w wersji 2.x, jeśli korzystamy z klas w nowym stylu z wersji 3.x) utworzona przez nas przed chwilą alternatywna wersja klasy Manager, zapisana w pliku *person-composite.py*, który utworzyliśmy w tym rozdziale, nie będzie w stanie przechwytywać i delegować atrybutów metod przeciążania operatorów takich jak `__repr__` bez ponownego ich zdefiniowania. Choć wiemy, że `__repr__` jest jedyną taką metodą wykorzystaną w naszym przykładzie, jest to ogólny problem klas opartych na delegacji.

Warto przypomnieć, że operacje wbudowane, takie jak wyświetlanie i indeksowanie, w niejawnym sposób wywołują metody przeciążania operatorów, takie jak `__repr__` czy `__add__`. W nowych klasach z Pythona 3.x takie wbudowane operacje nie przekierowują niejawnego pobierania atrybutów za pośrednictwem ogólnych menedżerów atrybutów — nie zostanie wywołana ani metoda `__getattr__` (wykonywana w przypadku niezdefiniowanych atrybutów), ani jej bliska kuzynka `__getattribute__` (wykonywana dla wszystkich atrybutów). Dlatego właśnie musimy ponownie zdefiniować metodę `__repr__` w alternatywnej klasie Manager, aby upewnić się, że po wykonaniu kodu w Pythonie 3.x wyświetlanie zostanie przekierowane do osadzonego obiektu Person.

Oznacz jako komentarz kod tej metody, aby zobaczyć to na żywo — instancja klasy Manager w wersji 3.x będzie wyświetlała dane w sposób domyślny, ale w wersji 2.x nadal użyje metody `__repr__` z klasy Person. W rzeczywistości metoda `__repr__` w klasie Manager nie jest w ogóle wymagana dla wersji 2.x, ponieważ została napisana tak, aby w wersji 2.x korzystać z normalnych, domyślnych klas (nazywanych również *klasycznymi*):

```
c:\code> py -3 person-composite.py
[Person: Robert Zielony, 0]
...etc...
<__main__.Manager object at 0x00000000029AA8D0>
c:\code> py -2 person-composite.py
[Person: Robert Zielony, 0]
...etc...
[Person: Tomasz Czarny, 60000]
```

Z technicznego punktu widzenia dzieje się tak, ponieważ wbudowane operacje rozpoczynają swoje niejawne poszukiwanie nazw metod w *instancjach* domyślnych, *klasycznych* klas z wersji 2.x, ale w klasach w *nowym stylu* z wersji 3.x zaczynają od

klas, całkowicie pomijając instancje. W przeciwnieństwie do tego jawne pobieranie atrybutów według nazwy zawsze jest najpierw kierowane do instancji w obu modelach. W klasycznych klasach wersji 2.x wbudowane funkcje przekierowują atrybuty również w ten sam sposób — wyświetlanie przekierowuje na przykład `__repr__` za pośrednictwem `__getattr__`. Dlatego oznaczenie jako komentarz metody `__repr__` klasy Manager nie działa w wersji 2.x: wywołanie jest delegowane do klasy Person. Klasa w nowym stylu dziedziczą również domyślne zachowanie `__repr__` z klasy nadzędnej `object`, które przysłaniałoby `__getattribute__`, ale nowego typu metoda `__getattribute__` nie przechwytyuje tej nazwy.

Jest to zmiana, jednak nie uniemożliwia ona działania kodu. Klasa nowego typu, oparte na delegacji mogą ogólnie redefiniować metody przeciążania operatorów w celu wydelegowania ich do opakowanych obiektów w Pythonie 3.0 — albo ręcznie, albo za pomocą narzędzi bądź klas nadzędnych. Zagadnienie to jest jednak zbyt zaawansowane, by poświęcać mu teraz więcej miejsca, dlatego nie będziemy się nim zajmować zbyt szczegółowo. Powrócimy do niego w rozdziałach 31. i 32. (w tym ostatnim formalnie zdefiniujemy klasy nowego typu), zobrazujemy je w przykładach zarządzania atrybutami w rozdziale 38. i używania dekoratorów klasy prywatnej w rozdziale 39. i pokażemy jako szczególny czynnik w niemal formalnej definicji *dziedziczenia* w rozdziale 40. W języku takim jak Python, który obsługuje zarówno przechwytywanie atrybutów, jak i przeciążanie operatorów, skutki tej zmiany mogą być bardzo szerokie.

Po uruchomieniu metoda `showAll` klasy `Department` wyświetla listę wszystkich zawartych w nim obiektów po zaktualizowaniu ich stanu w prawdziwie polimorficzny sposób za pomocą metody `giveRaises`:

```
[Person: Robert Zielony, 0]  
[Person: Anna Czerwona, 110000]  
[Person: Tomasz Czarny, 60000]
```

Co ciekawe, powyższy kod wykorzystuje *zarówno dziedziczenie, jak i kompozycję*. `Department` jest kompozytem osadzającym i kontrolującym inne obiekty do zagregowania, natomiast same osadzone obiekty `Person` oraz `Manager` wykorzystują dziedziczenie w celu dostosowania do własnych potrzeb. W innym przykładzie graficzny interfejs użytkownika może wykorzystywać *dziedziczenie* do dostosowania działania lub wyglądu etykiet i przycisków do własnych potrzeb, ale także *kompozycję* w celu utworzenia większych pakietów osadzonych widgetów, takich jak formularze do wprowadzania danych, kalkulatory czy edytory tekstu. Wykorzystywana struktura klas uzależniona jest od obiektów, których model próbujemy stworzyć — w rzeczywistości zdolność do modelowania bytów świata rzeczywistego w ten sposób jest jedną z mocnych stron programowania zorientowanego obiektowo.

Zagadnienia związane z projektowaniem, takie jak kompozycja, omówione są w rozdziale 31., dlatego odłożymy ich dalsze omawianie na później. Jednak znów w kontekście podstawowych opcji programowania zorientowanego obiektowo w Pythonie klasy `Person` i `Manager` wykorzystują już wszystko. Po opanowaniu postaw programowania zorientowanego obiektowo tworzenie uniwersalnych narzędzi służących do zastosowania go w łatwiejszy sposób w skryptach jest często naturalnym kolejnym krokiem — a także tematem kolejnego podrozdziału.

Krok 6. — wykorzystywanie narzędzi do introspekcji

Wykonajmy jeszcze jedną, ostatnią zmianę, zanim wrzucimy nasze obiekty do bazy danych. W obecnej postaci nasze klasy są kompletne i demonstrują większość podstaw programowania

zorientowanego obiektowo w Pythonie. Nadal jednak wiążą się z nimi dwa problemy, które najprawdopodobniej powinniśmy rozwiązać, zanim się do nich niepotrzebnie przyzwyczaimy:

- Po pierwsze, jeśli przyjrzymy się wyświetlaniu obiektów w obecnej postaci, zauważymy, że kiedy wyświetlamy obiekt `tom`, klasa `Manager` podpisuje go jako `Person`. Nie jest to niepoprawne z technicznego punktu widzenia, ponieważ `Manager` jest rodzajem wyspecjalizowanego obiektu `Person`. Mimo to lepiej byłoby wyświetlać obiekty z najbardziej konkretną (*najniższą* w drzewie) klasą, czyli klasą, na bazie której ten obiekt został utworzony.
- Po drugie, i chyba ważniejsze, obecny format wyświetlania pokazuje tylko atrybuty podane w naszej metodzie `__repr__`, a to może nie obejmować naszych przyszłych zmian. Nie możemy na przykład jeszcze zweryfikować, czy nazwa stanowiska obiektu `tom` została poprawnie ustawiona na '`manager`' przez konstruktor klasy `Manager`, ponieważ metoda `__repr__` klasy `Person` nie wyświetla tego pola. Co gorsza, jeśli kiedykolwiek rozszerzymy lub w inny sposób zmodyfikujemy zbiór atrybutów przypisanych do naszych obiektów w metodzie `__init__`, będziemy musieli pamiętać, by uaktualnić również metodę `__repr__` nowymi nazwami do wyświetlenia, gdyż inaczej z czasem metody te się rozsynchonizują.

Ostatni punkt oznacza, że znowu zastawiliśmy na siebie pułapkę potencjalnej dodatkowej pracy w przyszłości, wprowadzając *powtarzalność* w kodzie. Ponieważ rozbieżność w metodzie `__repr__` zostanie odzwierciedlona w danych wyjściowych programu, ta powtarzalność kodu może być jeszcze bardziej oczywista niż inne jej postaci, którymi zajmowaliśmy się wcześniej. Ponadto możliwość uniknięcia niepotrzebnej, dodatkowej pracy w przyszłości jest zazwyczaj bardzo pozytywnym czynnikiem.

Specjalne atrybuty klas

Oba problemy możemy rozwiązać za pomocą *narzędzi introspekcji* Pythona — specjalnych atrybutów oraz funkcji, które dają nam dostęp do pewnych mechanizmów wewnętrznych implementacji obiektów. Narzędzia te są nieco bardziej zaawansowane i zazwyczaj wykorzystywane są raczej przez osoby tworzące narzędzia dla innych programistów niż przez samych programistów piszących aplikacje. Mimo to podstawowa znajomość niektórych z tych narzędzi będzie przydatna, ponieważ pozwalają nam one pisać kod przetwarzający klasy na uniwersalne sposoby. W naszym kodzie znajdują się na przykład dwa punkty zaczepienia, które mogą nam pomóc — wprowadzone pod koniec poprzedniego rozdziału i używane we wcześniejszych przykładach:

- Wbudowany atrybut `instancja.__class__` udostępnia łącze z instancją do klasy, z której została ona utworzona. Klasy z kolei mają atrybut `__name__`, tak jak moduły, oraz sekwencję `__bases__` dającą dostęp do klas nadrzędnych. Możemy z tego skorzystać tutaj w celu wyświetlenia nazwy klasy, z której utworzono instancję, zamiast nazwy wpisanej na stałe do kodu.
- Wbudowany atrybut `obiekt.__dict__` udostępnia słownik z parami klucz-wartość dla każdego atrybutu dołączonego do obiektu przestrzeni nazw (w tym modułów, klas oraz instancji). Ponieważ jest to słownik, możemy między innymi pobierać jego listy kluczy, indeksować go po kluczu czy wykonywać iterację po kluczach w celu ogólnego przetworzenia wszystkich atrybutów. Możemy z tego skorzystać tutaj, by wyświetlić każdy atrybut instancji, a nie tylko te zapisane na stałe w naszym kodzie wyświetlającym, podobnie jak to robiliśmy w narzędziach modułów w rozdziale 25.

Pierwszą z tych kategorii poznaliśmy w poprzednim rozdziale, a poniżej zamieszczamy krótki przykład interaktywnej sesji Pythona z najnowszymi wersjami naszych klas z pliku `person.py`. Zwróci uwagę, w jaki sposób używamy w sesji interaktywnej instrukcji `from` do załadowania klasy `Person` — definicje klas znajdują się w modułach i są z nich importowane dokładnie tak samo, jak nazwy funkcji i inne zmienne:

```

>>> from person import Person
>>> bob = Person('Robert Zielony')
>>> bob
          # Uruchamia metodę __repr__
obiektu bob (a nie metodę __str__)
[Person: Robert Zielony, 0]

>>> print(bob)
          # podobnie: print => __str__ lub
__repr__

[Person: Robert Zielony, 0]

>>> bob.__class__
          # Pokazanie klasy obiektu bob i
jej nazwy
<class 'person.Person'>

>>> bob.__class__.__name__
'Person'

>>> list(bob.__dict__.keys())
          # Atrybuty są tak naprawdę
kluczami słownika
['pay', 'job', 'name']
          # By wymusić listę w wersji 3.x,
należy użyć list

>>> for key in bob.__dict__:
    print(key, '=>', bob.__dict__[key])  # Ręczne indeksowanie

pay => 0
job => None
name => Robert Zielony

>>> for key in bob.__dict__:
    print(key, '=>', getattr(bob, key))  # obiekt.atrybut, ale atrybut
jest zmienną

pay => 0
job => None
name => Robert Zielony

```

Jak wspominaliśmy w poprzednim rozdziale, niektóre atrybuty dostępne z instancji mogą nie być przechowywane w słowniku `__dict__`, jeżeli klasa instancji definiuje atrybut `__slots__` — opcjonalny i niezbyt znany atrybut klas w nowym stylu (a zatem wszystkich klas w Pythonie 3.x), który przechowuje atrybuty sekwencyjnie w instancji; może całkowicie wykluczać wystąpienie `__dict__`; powrócimy do tych zagadnień w rozdziałach 31. i 32. Ponieważ atrybuty `__slots__` naprawdę należą do klas zamiast do instancji i są rzadko używane, możemy je tutaj spokojnie zignorować i skupić się na zwykłym słowniku `__dict__`.

Skoro jednak tak zrobimy, pamiętaj, że niektóre programy muszą przechwytywać wyjątki spowodowane brakiem słownika `__dict__` lub używać metod `hasattr` do testowania atrybutów oraz `getattr` do ich pobierania, jeżeli wykorzystują atrybut `__slots__`. Jak zobaczymy w rozdziale 32., kod prezentowany w kolejnym podrozdziale nie zawiedzie, jeżeli zostanie użyty przez klasę z atrybutem `__slots__` (jego brak wystarcza, aby zagwarantować uzycie słownika `__dict__`), ale `__slots__` i inne „wirtualne” atrybuty nie będą raportowane jako dane instancji.

Uniwersalne narzędzie do wyświetlania

Interfejsy te możemy wykorzystać w klasie nadzędnej wyświetlającej dokładne nazwy klas i formatującą wszystkie atrybuty instancji dowolnej klasy. W edytorze tekstu należy otworzyć nowy plik w celu zapisania w nim następującego kodu — będzie to nowy, niezależny moduł o nazwie `clastoools.py`, implementujący taką właśnie klasę. Ponieważ jej metoda przeciążania wyświetlania `__repr__` będzie korzystała z uniwersalnych narzędzi do introspekcji, będzie ona działała na *każdej instancji*, bez względu na jej zbiór atrybutów. A ponieważ będzie klasą, automatycznie stanie się uniwersalnym narzędziem do formatowania — dzięki dziedziczeniu można ją wzmieszać w *dowolną klasę*, która chce skorzystać z jej formatu wyświetlania. Jako dodatkowy bonus, jeśli kiedykolwiek będziemy chcieli zmodyfikować sposób wyświetlania instancji, będziemy musieli zmienić jedynie tę klasę, gdyż każda klasa dziedzicząca jej metodę `__repr__` automatycznie pobierze nowy format przy następnym wykonywaniu.

```
# Nowy plik classtools.py

"Wybrane narzędzia do obsługi klas"

class AttrDisplay:

    """
    Udostępnia dziedziczoną metodę przeciążania wyświetlania, która pokazuje
    instancje z ich nazwami klas, a także parę
        nazwa=wartość dla każdego atrybutu przechowanego w samej instancji (ale
        nie atrybutów odziedziczonych po klasach).

    Można ją wzmieszać w dowolną klasę i będzie działała na dowolnej instancji.
    """

    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)

    def __repr__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())

if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
    class SubTest(TopTest):
        pass
    X, Y = TopTest(), SubTest()      # Utwórz dwie instancje
```

```

print(X)                                # Pokazanie wszystkich atrybutów
instancji

print(Y)                                # Pokazanie najniższej nazwy klasy

```

Warto zwrócić tutaj uwagę na notki dokumentacyjne. Ponieważ jest to narzędzie ogólnego przeznaczenia, warto dodać nieco dokumentacji, aby potencjalni użytkownicy mogli z niej skorzystać. Jak widzieliśmy w rozdziale 15., notki dokumentacyjne można umieszczać na początku definicji prostych funkcji i modułów, a także na początku definicji klas i ich metod. Funkcja `help` oraz narzędzie PyDoc pobierają i wyświetlają je automatycznie (notkom dokumentacyjnym przyjrzymy się ponownie w rozdziale 29.).

Po bezpośrednim uruchomieniu kod autotestu modułu tworzy dwie instancje i wyświetla je. Zdefiniowana tutaj metoda `__repr__` pokazuje klasę instancji oraz wszystkie jej nazwy i wartości atrybutów w kolejności zgodnej z posortowanymi nazwami atrybutów. Wyniki działania są identyczne w obu wersjach Pythona, 2.x i 3.x, ponieważ dane każdego z obiektów są generowane w postaci jednego ciągu znaków, zawierającego wszystkie potrzebne informacje:

```

C:\code> classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]

```

Kolejna uwaga dotycząca projektu: ponieważ nasza klasa używa metody `__repr__` zamiast `__str__`, jej mechanizm wyświetlania jest używany we wszystkich kontekstach, ale jej klienci nie będą mieli możliwości zapewnienia alternatywnego sposobu wyświetlania niskopoziomowego — nadal mogą co prawda dodawać metodę `__str__`, ale będzie ona dotyczyła tylko funkcji `print` i `str`. W bardziej ogólnych narzędziach użycie metody `__str__` ogranicza zasięg wyświetlania, ale pozostawia klientom możliwość dodania metody `__repr__` jako dodatkowego wyświetlacza dla sesji interaktywnej i danych zagnieżdżonych. Będziemy postępować zgodnie z tymi regułami podczas kodowania rozszerzonej wersji tej klasy w rozdziale 31., a w tym przykładzie będziemy trzymać się metody `__repr__`.

Atrybuty instancji a atrybuty klas

Osoby, które wystarczająco długo przyglądały się kodowi autotestu, zauważły zapewne, że jego klasa wyświetla tylko *atrybuty instancji*, dołączone do obiektu `self` na dole drzewa dziedziczenia — właśnie to zawiera słownik `__dict__` obiektu `self`. Zamierzoną konsekwencją jest zatem to, że nie widzimy atrybutów odziedziczonych przez instancje po klasach znajdujących się wyżej w drzewie (na przykład atrybutu `count`, wykorzystywanego w kodzie autotestu). Odziedziczone atrybuty klas dołączane są jedynie do klas i nie są kopowane do instancji.

Jeżeli jednak kiedykolwiek będziemy chcieli dołączyć wyświetlanie odziedziczonych atrybutów, możemy za pomocą łącza `__class__` przejść do klasy instancji, tam użyć słownika `__dict__` w celu pobrania atrybutów klasy, a następnie wykonać iterację po atrybutach `__bases__` klasy w celu wspięcia się do jeszcze wyższych klas nadrzędnych (powtarzając to w miarę potrzeby). Jeżeli jesteś fanem prostego kodu, wywołanie wbudowanej funkcji `dir` będzie miało ten sam efekt, ponieważ `dir` uwzględnia w posortowanej liście wyników także zmienne odziedziczone. W Pythonie 2.7:

```

>>> from person import Person          # 2.x: keys to lista, dir pokazuje
mniej

>>> bob = Person('Robert Zielony')

>>> bob.__dict__.keys()              # Tylko atrybuty instancji
['pay', 'job', 'name']

```

```
>>> dir(bob)                                # plus odziedziczone atrybuty z
      klas
['__doc__', '__init__', '__module__', '__repr__', 'giveRaise', 'job',
 'lastName', 'name', 'pay']
```

Jeśli używasz Pythona 3.x, wyniki będą się różnić i mogą zawierać więcej elementów, niż mógłbyś się spodziewać; oto wyniki działania dla dwóch ostatnich instrukcji uruchomionych w wersji 3.3 Pythona (kolejność list kluczów w każdym wywołaniu może być inna):

```
>>> list(bob.__dict__.keys())                # W wersji 3.x keys jest widokiem,
      a nie listą
['name', 'job', 'pay']

>>> dir(bob)                                # Wyniki w wersji 3.x obejmują
      metody klas
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 ...pominięto 31 kolejnych atrybutów...
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'giveRaise', 'job', 'lastName', 'name', 'pay']
```

Kod i wyniki dla wersji 2.x i 3.x Pythona różnią się, ponieważ dict.keys z wersji 3.x nie jest listą, a funkcja dir w tej wersji zwraca dodatkowe atrybuty implementacyjne typu klasy. Technicznie rzecz biorąc, funkcja dir zwraca w Pythonie 3.x więcej informacji, ponieważ wszystkie klasy tej wersji są klasami w nowym stylu i dziedziczą po typie klasy duży zbiór nazw przeciążania operatorów. W praktyce zazwyczaj będziesz chciał odfiltrować większość nazw typu `_X_` z wyników działania funkcji dir, ponieważ są to wewnętrzne szczegóły implementacyjne, a nie coś, co zwykle chciałbyś wyświetlić.

```
>>> len(dir(bob))
31
>>> list(name for name in dir(bob) if not name.startswith('__'))
['giveRaise', 'job', 'lastName', 'name', 'pay']
```

Chcąc zaoszczędzić nieco miejsca, zostawimy na razie opcjonalne wyświetlanie atrybutów klas za pomocą przechodzenia w górę drzewa klas lub wywołania funkcji dir jako sugerowane ćwiczenia do samodzielnego wykonania. Więcej wskazówek dotyczących tych kwestii znajdziesz w programie przechodzącym w górę drzewa dziedziczenia `classtree.py` z rozdziału 29., a także w programach wymieniających atrybuty z pliku `lister.py` w rozdziale 31.

Nazwy w klasach narzędziowych

Ostatnia drobnostka: ponieważ nasza klasa AttrDisplay z modułu `classutils` jest uniwersalnym narzędziem zaprojektowanym z myślą o zastosowaniu w dowolnych klasach, musimy być świadomi ryzyka potencjalnych i niezamierzonych konfliktów nazw zmiennych z klasami klienta. W obecnej formie kodu zakładamy, że klasy klienta mogą korzystać z metod `__str__` oraz `gatherAttrs`, jednak druga z nich może dawać więcej, niż oczekuje tego klasa podzielona. Jeżeli w klasie podzielonej użytkownik nieświadomie zdefiniuje własną metodę `gatherAttrs`, zepsuje to działanie naszej klasy, ponieważ znajdująca się niżej wersja tej metody z klasy podzielonej zostanie użyta w miejsce naszej.

Aby się o tym przekonać, wystarczy dodać metodę `gatherAttrs` do klasy `TopTest` w kodzie autotestu. O ile nowa metoda nie będzie identyczna lub celowo nie będzie dostosowywać oryginału do własnych potrzeb, nasza klasa narzędziowa nie będzie już działała tak, jak zakładaliśmy — `self.gatherAttrs` z klasy `AttrDisplay` użyje nowej wersji tej metody z instancji klasy `TopTest`:

```
class TopTest(AttrDisplay):  
    ....  
    def gatherAttrs(self):          # Zastępuje metodę z klasy  
        AttrDisplay!  
        return 'Mielonka'
```

Niekoniecznie jest to czymś złym — czasami chcemy, by inne metody były dostępne dla klas podległych w celu bezpośredniego wywołania lub dostosowania do własnych potrzeb. Jeżeli jednak naprawdę chcieliśmy tylko udostępnić metodę `__repr__`, takie rozwiązanie nie jest idealne.

Aby zminimalizować szanse wystąpienia takich konfliktów między nazwami, programiści Pythona często poprzedzają metodę, która nie jest przeznaczona do użycia na zewnątrz, *pojedynczym znakiem „_”*. W naszym przypadku da nam to metodę `_gatherAttrs`. Nie jest to rozwiązanie idealne (co, jeżeli inna klasa także definiuje metodę `_gatherAttrs`?), ale zazwyczaj wystarczające i jest to popularna w Pythonie konwencja nazywania metod wewnętrznych dla klasy.

Lepszym i rzadziej wykorzystywanym rozwiązaniem byłoby użycie *dwóch znaków „_”* przed nazwą metody, jak w `__gatherAttrs`. Python automatycznie rozszerza takie nazwy o nazwę klasy zawierającej, przez co stają się one unikatowe z punktu widzenia mechanizmu wyszukiwania dziedziczenia. Opcja ta znana jest pod nazwą *pseudoprivatnych atrybutów klas*, które bardziej szczegółowo omówimy w rozdziale 31. i następnie wdrożymy w rozszerzonej wersji naszej klasy. Na razie będziemy udostępniać obie nasze metody.

Ostateczna postać naszych klas

By teraz użyć tego uniwersalnego narzędzia w naszych klasach, wystarczy zaimportować je z modułu, wzmieszać do naszej klasy najwyższego poziomu za pomocą dziedziczenia i pozbyć się bardziej uszczegółowionego kodu utworzonej wcześniej metody `__repr__`. Nowa metoda przeciążająca wyświetlanie będzie dziedziczona przez instancje klasy `Person`, a także klasy `Manager`. `Manager` otrzymuje metodę `__repr__` z klasy `Person`, która teraz uzyskuje ją z klasy `AttrDisplay` zapisanej w innym module. Ostateczna wersja naszego pliku `person.py` po zastosowaniu tych zmian wygląda następująco:

```
#Plik classtools.py (nowy)  
... tak jak to wyglądało wcześniej...  
  
# Plik person.py (wersja ostateczna)  
....  
  
Rejestruje i przetwarza informacje o pracownikach.  
Uruchom ten plik bezpośrednio, aby przetestować jego klasy.  
....  
  
from classtools import AttrDisplay          # Użycie uniwersalnego  
narzędzia do wyświetlania
```

```

class Person(AttrDisplay):                      # Na tym poziomie dołączamy
metode __repr__


"""
Tworzy i przetwarza rekordy osób
"""

def __init__(self, name, job=None, pay=0):
    self.name = name
    self.job = job
    self.pay = pay

def lastName(self):                           # Zakładamy, że nazwisko
jest na końcu

    return self.name.split()[-1]

def giveRaise(self, percent):                 # Procent musi się mieścić
między 0 a 1

    self.pay = int(self.pay * (1 + percent))

class Manager(Person):

"""
Dostosowana do własnych potrzeb klasa Person ze specjalnymi wymaganiami
"""

def __init__(self, name, pay):
    Person.__init__(self, name, 'manager', pay) # Nazwa zawodu jest
domyślna

def giveRaise(self, percent, bonus=.10):
    Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob)
    print(anna)
    print(bob.lastName(), anna.lastName())
    anna.giveRaise(.10)
    print(anna)
    tom = Manager('Tomasz Czarny', 50000)
    tom.giveRaise(.10)
    print(tom.lastName())
    print(tom)

```

Ponieważ jest to ostateczna wersja kodu, dodaliśmy kilka *komentarzy* dokumentujących naszą pracę — notki dokumentacyjne i zwykłe komentarze `#` dla mniejszych notatek, zgodnie z najlepszymi praktykami, a także *puste wiersze* między definicjami metod dla zwiększenia czytelności — ogólnie dobry wybór stylu, gdy kod klasy lub metody staje się bardziej rozbudowany (do tej pory unikałem takiego zapisu dla małych klas, głównie dla zaoszczędzenia miejsca i utrzymania bardziej zwarteego kodu).

Po uruchomieniu kodu widzimy teraz wszystkie atrybuty naszych obiektów, a nie tylko te zapisane na stałe w oryginalnej metodzie `__repr__`. Rozwiążany został także nasz ostatni problem — ponieważ `AttrDisplay` pobiera nazwy klas bezpośrednio z instancji `self`, a każdy obiekt pokazany jest z nazwą najbliższej (znajdującej się najniżej w drzewie) klasy — obiekt `tom` jest teraz wyświetlany jako `Manager`, a nie `Person` i możemy wreszcie zweryfikować, że jego nazwa stanowiska została poprawnie wypełniona przez konstruktor klasy `Manager`.

```
C:\code> person.py
[Person: job=None, name=Robert Zielony, pay=0]
[Person: job=programista, name=Anna Czerwona, pay=100000]
Zielony Czerwona
[Person: job=programista, name=Anna Czerwona, pay=110000]
Czarny
[Manager: job=manager, name=Tomasz Czarny, pay=60000]
```

Oto bardziej przydatny sposób wyświetlania, o jaki nam chodziło. Z odleglejszej perspektywy nasza klasa wyświetlająca atrybuty stała się *uniwersalnym narzędziem*, które możemy wzmieszać w dowolną klasę za pomocą dziedziczenia w celu wykorzystania zdefiniowanego przez nią formatu wyświetlania. Co więcej, wszystkie klienci automatycznie pobiorą przyszłe modyfikacje naszego narzędzia. W dalszej części książki spotkamy się z koncepcjami związanymi z klasami, które mają jeszcze większe możliwości — takimi jak dekoratory oraz metaklasy. Obok narzędzi do introspekcji Pythona pozwalają nam one pisać kod rozszerzający klasy i zarządzający nimi w ustrukturyzowany, łatwy do utrzymania w przyszłości sposób.

Krok 7. i ostatni — przechowywanie obiektów w bazie danych

W tym momencie nasza praca jest prawie gotowa. Mamy teraz *system składający się z dwóch modułów*, który nie tylko implementuje nasze oryginalne cele projektowe, czyli reprezentowanie osób, ale także udostępnia uniwersalne narzędzie do wyświetlania atrybutów, które możemy w przyszłości wykorzystać w innych programach. Umieszczając kod funkcji oraz klas w plikach modułów, uzyskaliśmy to, że w naturalny sposób obsługują one ponowne wykorzystanie kodu. A zapisując program w postaci klas, uzyskaliśmy naturalną obsługę rozszerzania.

Choć jednak nasze klasy działają zgodnie z planem, tworzone przez nie obiekty nie są prawdziwymi rekordami bazy danych. Oznacza to, że jeśli wyłączymy Pythona, nasze instancje znikną — są one tymczasowymi obiektami w pamięci i nie są przechowywane w bardziej trwałym medium, takim jak plik, dlatego nie będą dostępne dla przyszłych wykonień programu. Okazuje się, że uczynienie obiektów instancji bardziej trwałymi jest bardzo proste — dzięki opcji Pythona znanej pod nazwą *trwałość obiektów* (ang. *object persistence*). W ten sposób obiekty będą istniały po zakończeniu działania programu, który je utworzył. W ostatnim kroku niniejszego przykładu sprawimy, że nasze obiekty będą prawdziwie trwałe.

Obiekty pickle i shelve

Trwałość obiektów zaimplementowana została za pomocą trzech modułów biblioteki standardowej, dostępnych w każdej wersji Pythona:

pickle

Serializuje dowolne obiekty Pythona na łańcuchy bajtowe i odwrotnie.

dbm (w Pythonie 2.x nosi nazwę anydbm)

Implementuje system plików dostępu według klucza, służący do przechowywania łańcuchów znaków.

shelve

Wykorzystuje dwa inne moduły w celu przechowania w pliku obiektów Pythona — dostępnych po kluczu.

Z modułami tymi spotkaliśmy się krótko w rozdziale 9., kiedy omawialiśmy podstawy plików. Udostępniają one opcje przechowywania danych o sporych możliwościach. Choć nie jesteśmy w stanie w pełni przedstawić ich zastosowań w tym przykładzie czy książce, są one na tyle proste, że krótkie wprowadzenie wystarczy, by zacząć z nimi pracę.

Moduł pickle

Moduł `pickle` jest rodzajem bardzo uniwersalnego narzędzia do formatowania i deformatowania obiektów. Po podaniu prawie dowolnego obiektu Pythona z pamięci jest na tyle sprytny, by potrafić przekształcić ten obiekt na łańcuch bajtów, który później można wykorzystać do odtworzenia oryginalnego obiektu w pamięci. Moduł `pickle` obsługuje prawie każdy obiekt, jaki możemy utworzyć — listy, słowniki, ich zagnieżdżone kombinacje i instancje klasy. Szczególnie użytecznymi obiektami dla serializacji są zwłaszcza te ostatnie, ponieważ udostępniają one zarówno dane (atrybuty), jak i działania (metody). Tak naprawdę kombinacja ta jest mniej więcej odpowiednikiem „rekordów” i „programów”. Ponieważ moduł `pickle` jest tak uniwersalny, może zastąpić dodatkowy kod, który w innym przypadku musielibyśmy napisać w celu utworzenia i przeanalizowania składniowo własnych reprezentacji pliku tekstowego dla naszych obiektów. Przechowując łańcuch serializacji w pliku, w rezultacie czynimy go trwałym. Później wystarczy załadować obiekt i zdeserializować go w celu jego odtworzenia.

Moduł shelve

Choć użycie samego modułu `pickle` w celu przechowania obiektów w zwykłych, płaskich plikach i późniejszego ich załadowania jest proste, moduł `shelve` udostępnia dodatkową warstwę struktury, która pozwala przechowywać zserializowane obiekty po *kluczu*. Moduł `shelve` przekłada obiekt na jego zserializowany za pomocą `pickle` łańcuch, a następnie przechowuje ten łańcuch pod kluczem w pliku `dbm`. Kiedy później go ładujemy, `shelve` pobiera zserializowany łańcuch po kluczu i odtwarza oryginalny obiekt w pamięci za pomocą `pickle`. To wszystko jest niezłą sztuczką, jednak dla naszego skryptu obiekty poddane serializacji wyglądają tak jak *słownik* — indeksuje się je po kluczu w celu pobrania, przypisuje do kluczy w celu przechowania i korzysta z narzędzi słowników, takich jak `len`, `in` czy `dict.keys`, w celu otrzymania informacji. Zserializowane za pomocą `shelve` obiekty automatycznie odwzorowują operacje na słownikach na obiekty przechowane w pliku.

Tak naprawdę dla naszego skryptu jedyna różnica w kodzie pomiędzy `shelve` a normalnym słownikiem polega na tym, że zserializowane za pomocą `shelve` obiekty musimy początkowo otwierać, a po wprowadzeniu zmian zamykać. Rezultat jest taki, że moduł `shelve` udostępnia prostą bazę danych służącą do przechowywania i pobierania własnych obiektów Pythona po kluczu, tym samym czyniąc je trwałymi pomiędzy wywołaniami programu. Nie obsługuje narzędzi tworzenia zapytań, takich jak SQL, i brakuje mu pewnych zaawansowanych opcji dostępnych w profesjonalnych bazach danych (takich jak prawdziwe przetwarzanie transakcji),

jednak własne obiekty Pythona przechowane za pomocą `shelve` można przetwarzać z pełną mocą języka Python po uprzednim pobraniu ich z powrotem za pomocą klucza.

Przechowywanie obiektów w bazie danych za pomocą `shelve`

Korzystanie z modułów `pickle` oraz `shelve` jest zagadnieniem stosunkowo zaawansowanym, dlatego nie będziemy się tutaj zagłębiać we wszystkie szczegóły. Więcej informacji na ich temat można znaleźć w dokumentacji biblioteki standardowej, a także książkach skupiających się na aplikacjach, takich jak *Programming Python*. Wszystko to jednak jest łatwiejsze w kodzie Pythona niż tekstowym opisie, dlatego zabierzmy się za kod.

Napiszemy nowy skrypt, który serializuje obiekty naszych klas za pomocą modułu `shelve`. W edytorze tekstu należy otworzyć nowy plik, który nazwiemy `makedb.py`. Ponieważ jest to nowy plik, będziemy musieli zainportować nasze klasy w celu utworzenia kilku instancji do przechowania. Wcześniej, w sesji interaktywnej, importowaliśmy klasę za pomocą instrukcji `from`, jednak tak jak w przypadku funkcji i innych zmiennych, zawsze istnieją dwa sposoby załadowania z pliku (nazwy klas są takimi samymi zmiennymi jak wszystkie pozostałe i w tym kontekście nie ma nic magicznego):

```
import person                                # Załadowanie klasy za pomocą instrukcji
import

bob = person.Person(...)                      # Przejście nazwy modułu

from person import Person                     # Załadowanie klasy za pomocą instrukcji
from

bob = Person(...)                            # Bezpośrednie użycie zmiennej
```

W celu załadowania klasy w naszym skrypcie skorzystamy z instrukcji `from`, jednak zrobimy tak tylko dla tego, że wymaga to mniej pisania. Aby ułatwić sobie życie, możesz skopiować lub ponownie wpisać ten kod w celu utworzenia w nowym skrypcie instancji naszych klas, abyśmy mieli co przechowywać (to tylko prosta demonstracja, więc nie będziemy się tu przejmować powtarzalnością kodu autotestu). Kiedy mamy już instancje, przechowanie ich za pomocą `shelve` jest prawie trywialne. Po prostu importujemy moduł `shelve`, otwieramy nowy obiekt `shelve` z nazwą pliku zewnętrznego, przypisujemy w nim obiekty do kluczy, a następnie po zakończeniu zamykamy obiekt `shelve`, ponieważ wprowadziliśmy zmiany.

```
# Plik makedb.py: przechowanie obiektów klasy Person w bazie danych modułu
shelve

from person import Person, Manager              # Załadowanie naszych klas

bob = Person('Robert Zielony')                 # Ponowne utworzenie obiektów
do przechowania

anna = Person('Anna Czerwona', job='programista', pay=100000)

tom = Manager('Tomasz Czarny', 50000)

import shelve

db = shelve.open('persondb')                    # Nazwa pliku, w którym
przechowywane są obiekty

for obj in (bob, anna, tom):                   # Użycie atrybutu name obiektu
jako klucza
```

```

        db[obj.name] = obj
    shelf po kluczu

    db.close()                                # Przechowanie obiektu w pliku
    zmian

```

Warto zwrócić uwagę na to, w jaki sposób przypisujemy obiekty do obiektu `shelve`, wykorzystując do tego ich własne atrybuty `name` w postaci kluczy. Robimy to dla wygody — w obiekcie `shelve` kluczem może być dowolny łańcuch znaków, w tym taki, który utworzymy jako unikalny za pomocą narzędzi takich, jak identyfikator procesu czy data i godzina (dostępnych w modułach biblioteki standardowej `os` oraz `time`). Jedyna zasada jest taka, że klucze muszą być łańcuchami znaków i powinny być unikalne, ponieważ możemy przechować tylko jeden obiekt na klucz (choć obiekt ten może być listą, słownikiem lub innym obiektem zawierającym wiele obiektów).

Wartości przechowywane pod kluczami mogą jednak być prawie dowolnego rodzaju obiektami Pythona — typami wbudowanymi, jak łańcuchy znaków, listy czy słowniki, a także instancjami klas zdefiniowanych przez użytkownika i zagnieżdżonymi kombinacjami wszystkich tych kategorii. Na przykład atrybuty `name` i `job` naszych obiektów mogą być zagnieżdżone w słownikach i listach, tak jak to pokazywaliśmy we wcześniejszych wydaniach tej książki (choć takie rozwiązanie wymagałoby pewnego przeprojektowania obecnego kodu).

I to tyle — jeżeli skrypt nie zwraca żadnych danych wyjściowych po wykonaniu, oznacza to najprawdopodobniej, że zadziałał. Nic nie wyświetlamy, a jedynie tworzymy i przechowujemy obiekty w plikach bazy danych.

```
C:\code> makedb.py
```

Interaktywna eksploracja obiektów `shelve`

W tym momencie w katalogu bieżącym mamy jeden lub kilka prawdziwych plików, których nazwy rozpoczynają się od „`persondb`”. Same tworzone pliki mogą się różnić dla poszczególnych platform i, tak jak funkcja wbudowana `open`, nazwa pliku w `shelve.open()` podawana jest względem aktualnego katalogu roboczego, o ile nie zawiera ścieżki do katalogu. Bez względu na miejsce przechowania pliki te implementują plik z dostępem po kluczu, który zawiera zserializowaną reprezentację naszych trzech obiektów Pythona. Nie należy usuwać tych plików — są one naszą bazą danych i będziemy je musieli skopiować lub przenieść, kiedy będziemy tworzyć kopię zapasową bazy.

Można sprawdzić zawartość plików `shelve`, jeżeli ktoś ma na to ochotę — albo z poziomu Eksploratora Windows, albo powłoki Pythona, jednak są to pliki binarne, a większa część ich zawartości ma niewielki sens poza kontekstem modułu `shelve`. W Pythonie 3.x i bez zainstalowanego żadnego dodatkowego oprogramowania nasza baza danych przechowana zostaje w trzech plikach (w wersji 2.x będzie to tylko jeden plik, `persondb`, ponieważ moduł rozszerzenia `bsddb` jest w Pythonie instalowany automatycznie dla plików `shelve`; w wersji 3.x `bsddb` jest dodatkiem zewnętrznym na licencji `open source`).

Na przykład standardowy moduł globalnej biblioteki Pythona pozwala z poziomu kodu wyświetlić listę plików w celu weryfikacji zawartości katalogu, a następnie otwierać pliki w trybie tekstowym lub binarnym, aby eksplorować ciągi znaków i bajtów przechowywane w tych plikach:

```

>>> import glob
>>> glob.glob('person*')
['person-composite.py', 'person-department.py', 'person.py', 'person.pyc',
'persondb.bak', 'persondb.dat', 'persondb.dir']

```

```

>>> print(open('persondb.dir').read())
'Anna Czerwona', (512, 92)
'Tomasz Czarny', (1024, 91)
'Robert Zielony', (0, 80)
>>> print(open('persondb.dat', 'rb').read())
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x03\x00\x00\x00jobq\x03NX\x03\x00
...reszta zawartości pominięta...

```

Zawartość tą da się odszyfrować, jednak może ona być różna dla różnych platform i trudno to uznać za przyjazny dla użytkownika interfejs bazy danych! W celu lepszego zweryfikowania naszej pracy możemy napisać kolejny skrypt albo pobawić się plikiem `shelve` w sesji interaktywnej. Ponieważ obiekty `shelve` są obiektami Pythona zawierającymi inne obiekty Pythona, możemy je przetwarzanie za pomocą normalnej składni i trybów programowania tego języka. Poniżej sesja interaktywna staje się *klientem bazy danych*:

```

>>> import shelve
>>> db = shelve.open('persondb')                                # Ponowne otwarcie
pliku shelve
>>> len(db)                                                    # Przechowano 3
„rekordy”
3
>>> list(db.keys())                                         # keys w celu
zindeksowania
['Anna Czerwona', 'Tomasz Czarny', 'Robert Zielony']      # w wersji 3.x
używamy funkcji list() do uzyskania listy
>>> bob = db['Robert Zielony']                               # Pobranie obiektu
bob po kluczu
>>> bob                                                       # Wywołuje metodę
__repr__ z klasy AttrDisplay
[Person: job=None, name=Robert Zielony, pay=0]
>>> bob.lastName()                                         # Wykonuje lastName z
klasy Person
'Zielony'
>>> for key in db:                                           # Iteracja, pobranie,
wyświetlenie
    print(key, '=>', db[key])
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=100000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
>>> for key in sorted(db):

```

```

print(key, '=>', db[key])                                # Iteracja po
posortowanych kluczach

Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=100000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

```

Warto zauważyc, że nie musimy tutaj importować naszych klas Person czy Manager w celu załadowania lub wykorzystania przechowywanych obiektów. Możemy na przykład swobodnie wywołać metodę `lastName` obiektu bob i automatycznie otrzymać własny format wyświetlania, nawet jeśli jego klasa Person nie znajduje się tutaj w naszym zakresie. Dzieje się tak, ponieważ gdy Python serializuje instancję klasy, zapisuje atrybuty `self` tej instancji, a także nazwę klasy, z której została ona utworzona, oraz modułu, w którym klasa ta się znajduje. Kiedy później pobieramy obiekt bob z pliku `shelve` i poddajemy deserializacji, Python automatycznie ponownie importuje klasę i łączy z nią obiekt bob.

Zaletą tego rozwiązania jest to, że instancje klas automatycznie uzyskują wszystkie zachowania swoich klas, kiedy w przyszłości zostają załadowane. Klasy musimy importować tylko w celu utworzenia nowych instancji, a nie przetwarzania istniejących. Choć jest to celowe rozwiązanie, ma ono nieco mieszane konsekwencje:

- *Wadą* jest to, że klasy i ich pliki modułów trzeba mówić importować, kiedy instancja zostanie później załadowana. Z formalnego punktu widzenia klasy podatne na serializację muszą być zapisane w kodzie na najwyższym poziomie pliku modułu dostępnego z katalogu wymienionego w ścieżce wyszukiwania modułów `sys.path` (i nie powinny się znajdować w metodzie `__main__` modułu większości skryptów, o ile przy użyciu nie znajdują się one zawsze w tym module). Z powodu takich wymagań w zakresie zewnętrznych plików modułów niektóre aplikacje decydują się serializować prostsze obiekty, takie jak słowniki czy listy, zwłaszcza gdy mają one zostać przetransferowane za pomocą internetu.
- *Zaletą* jest to, że modyfikacje w pliku z kodem źródłowym klasy są pobierane automatycznie, gdy instancje klasy zostaną ponownie załadowane. Często nie ma potrzeby uaktualniać samych przechowywanych obiektów, ponieważ uaktualnienie kodu ich klasy modyfikuje ich działanie.

Obiekty `shelve` mają także dobrze znane ograniczenia (o kilku z nich wspomnieliśmy na końcu niniejszego rozdziału przy omawianiu sugerowanych baz danych). W przypadku prostego przechowywania obiektów moduły `shelve` i `pickle` są jednak niezwykle łatwymi w użyciu narzędziami.

Uaktualnianie obiektów w pliku `shelve`

Czas na ostatni skrypt. Napiszemy program uaktualniający instancję (rekord) z każdym wykonaniem w celu udowodnienia, że nasze obiekty są naprawdę *trwałe* (to znaczy ich wartości bieżące są dostępne z każdym wykonaniem programu Pythona). Poniższy plik, `updatedb.py`, wyświetla zawartość bazy danych i za każdym razem daje podwyżkę jednemu z naszych obiektów. Jeśli prześledzimy, co się w nim dzieje, zauważymy, że mnóstwo funkcjonalności otrzymujemy gratis — wyświetlanie obiektów automatycznie wykorzystuje ogólną metodę przeciążania operatorów `__repr__`, a podwyżki przyznaje się, wywołując napisaną wcześniej metodę `giveRaise`. Wszystko to „po prostu działa” dla obiektów opartych na modelu dziedziczenia z programowania zorientowanego obiektowo, nawet jeśli obiekty te znajdują się w pliku.

```

# Plik updatedb.py: uaktualnienie obiektu klasy Person w bazie danych

import shelve

```

```

db = shelve.open('persondb')                      # Ponowne otwarcie pliku shelve z
tą samą nazwą pliku

for key in sorted(db):                            # Iteracja w celu wyświetlenia
    obiektów bazy danych

        print(key, '\t=>', db[key])                # Wyświetlenie za pomocą własnego
formatu

anna = db['Anna Czerwona']                       # Indeksowanie za pomocą klucza w
celu pobrania

anna.giveRaise(.10)                                # Uaktualnienie w pamięci za
pomocą metody klasy

db['Anna Czerwona'] = anna                        # Przypisanie do klucza w celu
uaktualnienia w pliku shelve

db.close()                                         # Zamknięcie po wprowadzeniu zmian

```

Ponieważ powyższy skrypt wyświetla po uruchomieniu zawartość bazy danych, musimy wywołać go kilka razy, by zobaczyć, jak zmieniają się nasze obiekty. Poniżej widać jego działanie — wyświetlenie wszystkich rekordów i zwiększenie atrybutu `pay` obiektu `anna` z każdym wykonaniem (jak widać, dla obiektu `anna` jest to dość przyjemny skrypt...; może moglibyśmy go uruchamiać w regularnych odstępach czasu za pośrednictwem np. zadania cron?):

```

c:\code> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=100000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

c:\code> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=110000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

c:\code> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=121000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

c:\code> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=133100]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

```

I znów: to, co widzimy tutaj, jest rezultatem działania narzędzi `shelve` i `pickle` otrzymanych od Pythona oraz działań, które sami zapisaliśmy w kodzie naszych klas. Ponownie możemy zweryfikować w sesji interaktywnej, czy nasz skrypt działa (sesja będzie odpowiednikiem klienta bazy danych):

```

c:\code> python
>>> import shelve

```

```
>>> db = shelve.open('persondb')          # Ponowne otwarcie bazy danych
>>> rec = db['Anna Czerwona']           # Pobranie obiektu po kluczu
>>> rec
[Person: job=programista, name=Anna Czerwona, pay=146410]
>>> rec.lastName()
'Czerwona'
>>> rec.pay
146410
```

Kolejny przykład trwałości obiektów w tej książce można znaleźć w rozdziale 31., w ramce zatytułowanej „Warto pamiętać: klasy i trwałość obiektów”. Przechowujemy tam nieco większy obiekt kompozytowy w pliku za pomocą pickle, a nie shelve, jednak rezultat jest podobny. Więcej informacji na temat wykorzystywania modułów pickle i shelve oraz przykłady ich zastosowania możesz znaleźć w rozdziale 9. (podstawy pracy z plikami), rozdziale 37. (zmiany w obsłudze ciągów znaków w wersji 3.x), w innych książkach lub dokumentacji Pythona.

Przyszłe kierunki rozwoju

I to właściwie koniec niniejszego przykładu. Zaprezentowaliśmy wszystkie podstawy programowania zorientowanego obiektywem w Pythonie w akcji, nauczyliśmy się sposobów unikania powtarzalności kodu, a także — powiązanych z tą kwestią — problemów z utrzymywaniem kodu w przyszłości. Zbudowaliśmy klasy wykonujące prawdziwe zadania. Jako dodatkowy bonus zbudowaliśmy także prawdziwe rekordy bazy danych, przechowując je w pliku shelve Pythona, tak by informacje te były trwałe.

Moglibyśmy oczywiście zrobić jeszcze więcej. Przykładowo moglibyśmy rozszerzyć nasze klasy w taki sposób, by stały się bardziej realistyczne, czy dodać do nich nowe rodzaje zachowania. Dawanie podwyżki powinno na przykład sprawdzać, czy jej wysokość mieści się między zero a jeden — takie rozszerzenie dodamy, gdy w dalszej części książki spotkamy dekoratory. Możemy także przekształcić ten przykład w osobistą bazę danych kontaktów, modyfikując informacje o stanie przechowywane w obiektach, a także metody klas wykorzystywane do ich przetwarzania. Pozostawimy to jednak jako sugerowane ćwiczenie otwarte, w którym można w pełni wykorzystać swoją wyobraźnię.

Moglibyśmy także rozszerzyć zakres wykorzystywanych narzędzi wbudowanych w Pythona lub swobodnie dostępnych w świecie oprogramowania open source:

Graficzne interfejsy użytkownika

W obecnej postaci możemy jedynie przetwarzać naszą bazę danych za pomocą interfejsu wiersza poleceń oraz skryptów. Moglibyśmy również popracować nad rozszerzeniem użyteczności naszej bazy danych, dodając graficzny interfejs użytkownika służący do przeglądania i uaktualniania jej rekordów. GUI można budować w sposób przenośny za pomocą obsług modułu tkinter (Tkinter w wersji 2.x) biblioteki standardowej Pythona lub zewnętrznych pakietów narzędzi, takich jak WxPython czy PyQt. Moduł tkinter udostępniany wraz z Pythonem pozwala szybko budować proste graficzne interfejsy użytkownika i jest idealny do uczenia się technik programowania GUI. WxPython i PyQt są nieco bardziej skomplikowane w użyciu, ale często dają w rezultacie graficzny interfejs użytkownika wyższej klasy.

Strony internetowe

Choć graficzne interfejsy użytkownika są wygodne i szybkie, jeśli chodzi o dostępność — nic nie pobije internetu. Możemy zatem zaimplementować stronę internetową służącą do przeglądania i aktualniania rekordów zamiast (lub obok) GUI i sesji interaktywnej. Strony internetowe można tworzyć albo za pomocą prostych narzędzi do skryptów CGI wbudowanych w Pythona, albo rozbudowanych platform zewnętrznych, takich jak Django, TurboGears, Pylons, web2Py, Zope czy Google's App Engine. W Internecie dane nadal mogą być przechowywane w obiektach pickle, plikach shelve czy innym medium opartym na Pythonie. Skrypty przetwarzające dane są po prostu wykonywane automatycznie na serwerze w odpowiedzi na żądania z przeglądarki internetowej bądź innych klientów i zwracają kod HTML umożliwiający interakcję z użytkownikiem — albo bezpośrednią, albo za pomocą interfejsu dla API platform internetowych. Aplikacje RIA (ang. *Rich Internet Application*), takie jak Silverlight czy pyjs (dawniej *pyjamas*), również próbują połączyć interaktywność graficznych interfejsów użytkownika z wdrożeniem internetowym.

Usługi sieciowe

Choć klienci internetowe często potrafią analizować składniowo informacje pochodzące z odpowiedzi stron internetowych (technika ta znana jest pod nazwą *screen scraping*), możemy pójść o krok dalej i udostępnić bardziej bezpośredni sposób pobierania rekordów w internecie za pomocą interfejsu usług sieciowych, takiego jak SOAP czy wywołania XML-RPC — takie interfejsy API obsługiwane są albo przez samego Pythona, albo przez narzędzia zewnętrzne typu open source, które na potrzeby transmisji dokonują konwersji danych do formatu XML. W skryptach Pythona takie interfejsy API zwracają dane bardziej bezpośrednio niż w postaci tekstu osadzonego w kodzie HTML strony odpowiedzi.

Bazy danych

Jeżeli nasza baza danych ma większy rozmiar lub ma duże znaczenie, możemy przenieść ją z plików shelve do bardziej rozbudowanego mechanizmu przechowywania, takiego jak zorientowany obiektywnie system bazy danych ZODB lub bardziej tradycyjny system relacyjnej bazy danych oparty na SQL, taki jak MySQL, Oracle czy PostgreSQL. Sam Python udostępniany jest z wbudowanym systemem bazy danych SQLite, jednak inne rozwiązania na licencji open source są również łatwo dostępne. Baza ZODB jest na przykład nieco podobna do plików shelve Pythona, jednak likwiduje wiele z jego ograniczeń, lepiej obsługuje większe bazy danych, równolegle aktualizacje, przetwarzanie transakcji i automatyczne zapisywanie zmian w pamięci (shelves mogą buforować obiekty i w odpowiednim momencie zapisywać je na dysku z użyciem opcji writeback, ale ma to pewne ograniczenia: więcej szczegółowych informacji na ten temat znajdziesz w internecie lub innych źródłach). Systemy oparte na SQL, takie jak MySQL, oferują narzędzia klasy enterprise do przechowywania danych i mogą być używane bezpośrednio z poziomu skryptu Pythona. Jak widzieliśmy w rozdziale 9., baza MongoDB oferuje alternatywne podejście do przechowywania dokumentów JSON, które są mocno zbliżone do słowników i list Pythona, ale jednocześnie neutralne językowo, w przeciwieństwie do danych typu pickle.

Odwzorowanie obiektowo-relacyjne

Jeżeli przeniesiemy przechowywane dane do systemu relacyjnej bazy danych, nie musimy wcale rezygnować z narzędzi do programowania zorientowanego obiektywnie w Pythonie. Narzędzia do odwzorowania obiektowo-relacyjnego (ang. *object-relational mappers, ORM*), takie jak SQLAlchemy czy SQLObject, potrafią automatycznie odwzorowywać relacyjne tabele i wiersze na klasy oraz obiekty Pythona i odwrotnie, w taki sposób, byśmy mogli przetwarzać przechowane dane za pomocą normalnej składni klas Pythona. Takie rozwiązanie stanowi alternatywę dla zorientowanych obiektywnie systemów bazy danych, takich jak shelve i ZODB, i wykorzystuje możliwości zarówno relacyjnych baz danych, jak i modelu klas Pythona.

Choć mam nadzieję, że powyższe wprowadzenie zaostrzy apetyt Czytelników na dalsze zapoznanie się z nimi, wszystkie z przedstawionych powyżej zagadnień wykraczają znacznie

poza zakres tego przykładu i ogólnie książki. Osoby, które chcą samodzielnie się z nimi zapoznać, odsyłam do internetu, dokumentacji biblioteki standardowej Pythona, a także książek poświęconych dziedzinie aplikacji, takich jak *Programming Python*. W tej ostatniej pozycji kontynuuję przedstawiony tutaj przykład, pokazując, jak można dodać do bazy danych zarówno graficzny interfejs użytkownika, jak i stronę internetową, tak by można było przeglądać i aktualizować rekordy instancji. Mam nadzieję, że tam się spotkamy; najpierw jednak wróćmy do podstaw klas i skończmy omawianie zagadnień dotyczących podstaw języka Python.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy praktyczne aspekty działania wszystkich podstaw klas i programowania zorientowanego obiektowo w Pythonie, budując krok po kroku prosty, choć bardziej realistyczny przykład. Dodaliśmy konstruktory, metody, przeciążanie operatorów, dostosowywanie do własnych potrzeb za pomocą klas podległych, narzędzia introspekcji, a także przy okazji zapoznaliśmy się z innymi koncepcjami, takimi jak kompozycja, delegacja i polimorfizm.

Na koniec uczyniliśmy obiekty utworzone za pomocą klas trwałymi, przechowując je w bazie danych obiektu `shelve` — prostego w użyciu systemu służącego do zapisywania i pobierania własnych obiektów Pythona po kluczu. Omawiając podstawy klas, poznaliśmy także kilka sposobów fakturyzacji kodu w celu zmniejszenia jego powtarzalności i zminimalizowania kosztów utrzymania w przyszłości. Na koniec przyjrzelismy się krótko sposobom rozszerzania kodu za pomocą narzędzi do programowania aplikacji, takich jak graficzne interfejsy użytkownika i bazy danych, omówione w bardziej zaawansowanych publikacjach.

W kolejnych rozdziałach tej części książki powrócimy do omawiania szczegółów leżących u podstaw modelu klas Pythona i zapoznamy się z zastosowaniem w tym języku pewnych koncepcji projektowych wykorzystywanych w celu łączenia klas w większych programach. Zanim jednak przejdziemy dalej, czas wykonać quiz, by sprawdzić wiedzę na temat zagadnień omówionych w niniejszym rozdziale. Ponieważ wykonaliśmy w nim sporo praktycznej pracy, zakończymy rozdział zbiorem pytań teoretycznych, zaprojektowanych z myślą o prześledzeniu części utworzonego tu kodu i zastanowieniu się nad najważniejszymi koncepcjami leżącymi u jego podstaw.

Sprawdź swoją wiedzę — quiz

1. Skąd pochodzi logika formatowania wyświetlanego w sytuacji, gdy pobieramy obiekt klasy `Manager` z pliku `shelve` i wyświetlamy go?
2. Kiedy pobieramy obiekt klasy `Person` z pliku `shelve` bez importowania jego modułu, skąd obiekt ten wie, że ma metodę `giveRaise`, którą możemy wywołać?
3. Dlaczego tak ważne jest, by przetwarzanie umieszczać w metodach, zamiast zapisywać je na stałe poza klasami?
4. Dlaczego lepiej jest dostosowywać kod do własnych potrzeb za pomocą klas podległych, niż kopować oryginał i modyfikować go?
5. Dlaczego lepiej jest wywoływać metody klas nadległych w celu wykonania działań domyślnych, zamiast kopować je i modyfikować ich kod w klasach podległych?
6. Dlaczego lepiej jest korzystać z narzędzi takich jak `__dict__`, pozwalających na uniwersalne przetwarzanie obiektów, zamiast pisać własny kod dla każdego z typów

klas?

7. Mówiąc ogólnie, kiedy możemy zdecydować się na korzystanie z osadzania obiektów i kompozycji zamiast dziedziczenia?
8. Co należałoby zmienić, gdyby obiekty zakodowane w tym rozdziale używały słowników dla nazw i list dla zadań, jak w podobnych przykładach wcześniej w tej książce?
9. W jaki sposób można zmodyfikować klasy z niniejszego rozdziału, tak by implementowały one osobistą bazę danych kontaktów w Pythonie?

Sprawdź swoją wiedzę — odpowiedzi

1. W ostatecznej wersji naszych klas Manager dziedziczy swoją metodę wyświetlania `__repr__` po klasie `AttrDisplay` z oddzielnego modułu `classestools`, znajdującej się o dwa poziomy wyżej w hierarchii klas. Klasa Manager nie ma własnej metody, dlatego wyszukiwanie dziedziczenia wspina się do jej klasy nadzędnej Person. Ponieważ tam także nie ma metody `__repr__`, wyszukiwanie przechodzi jeszcze wyżej i odnajduje metodę w klasie `AttrDisplay`. Nazwy klas podane w nawiasach w wierszu nagłówka instrukcji `class` udostępniają łącza do znajdujących się wyżej klas nadzędnych.
2. Pliki `shelve` (a tak naprawdę wykorzystywany przez nie moduł `pickle`) automatycznie ponownie łączą instancję z klasą, z której została utworzona, kiedy instancja ta jest później z powrotem ładowana do pamięci. Python wewnętrznie ponownie importuje klasę z jej modułu, tworzy instancję z przechowanymi atrybutami i ustawia łącze `__class__` instancji, tak by wskazywało na oryginalną klasę. W ten sposób załadowane instancje automatycznie pozyskują wszystkie oryginalne metody (takie, jak `lastName`, `giveRaise` czy `__repr__`), nawet jeśli nie zaimportowaliśmy klasy instancji do naszego zasięgu.
3. Przesuwanie przetwarzania do metod jest ważne, ponieważ dzięki temu w przyszłości będziemy mieli tylko jedną kopię do zmodyfikowania, a metody mogą być wykonywane na dowolnej instancji. Na tym polega w Pythonie *hermetyzacja* — opakowywanie logiki w interfejsy, tak by lepiej obsługiwać utrzymywanie kodu w przyszłości. Jeśli tego nie zrobimy, narazimy się na powtarzalność kodu, co może znacznie zwiększyć wysiłek niezbędny w przyszłości do jego utrzymania w przypadku jego ewolucji.
4. Dostosowanie kodu do własnych potrzeb za pomocą klas podzędnych zmniejsza ilość wymaganej pracy programistycznej. W programowaniu zorientowanym obiektowo kod tworzymy za pomocą *dostosowywania* tego, co już zostało wykonane, do własnych potrzeb zamiast kopiowania czy modyfikowania istniejącego. To właśnie najważniejsza koncepcja leżąca u podstaw programowania zorientowanego obiektowo — ponieważ możemy z łatwością rozszerzać poprzednią pracę, tworząc kod nowych klas podzędnych, możemy wykorzystać to, co zrobiliśmy wcześniej. Jest to o wiele lepsze rozwiązanie od zaczynania za każdym razem od podstaw czy wprowadzania kilku powtarzających się kopii kodu, które w przyszłości będą musieli aktualniac.
5. Kopiowanie i modyfikowanie kodu *podwaja* potencjalny wysiłek wymagany w przyszłości, bez względu na kontekst. Jeśli klasa podzędna musi wykonać działania domyślne zapisane w kodzie klasy nadzędnej, o wiele lepiej jest wywołać je za pośrednictwem nazwy klasy nadzędnej, niż kopować ich kod. Tak samo jest

również w przypadku konstruktorów klas nadrzędnych. I znów, kopiowanie kodu powoduje jego powtarzalność, co staje się sporym problemem w miarę jego ewolucji.

6. Narzędzia uniwersalne pomagają uniknąć zapisywania w kodzie na stałe rozwiązań, które muszą pozostać zsynchronizowane z resztą klasy w miarę jej ewolucji w czasie. Uniwersalna metoda `__repr__` nie musi na przykład być uaktualniana za każdym razem, gdy w konstruktorze `__init__` do instancji dodawany jest nowy atrybut. Dodatkowo uniwersalna metoda `print` dziedziczoną przez wszystkie klasy pojawia się (i musi być modyfikowana) tylko w jednym miejscu — zmiany wprowadzone w wersji ogólnej są pobierane przez wszystkie klasy dziedziczące po klasie uniwersalnej. I znów, eliminacja powtarzalności kodu ogranicza wysiłek niezbędny w przyszłości. Jest to jedna z podstawowych zalet klas.
7. Dziedziczenie najlepiej nadaje się do tworzenia kodu rozszerzeń opartych na bezpośrednim dostosowywaniu do własnych potrzeb (jak nasza klasa `Manager`, będąca wyspecjalizowaną wersją klasy `Person`). Kompozycja dobrze sprawdza się w sytuacjach, gdy kilka obiektów agregowanych jest w całość i sterowanych przez klasę warstwy kontrolera. Dziedziczenie przekazuje wywołania w góre, w celu ponownego ich wykorzystania, natomiast kompozycja przekazuje je w dół, w celu delegowania. Dziedziczenie i kompozycja nie wykluczają się wzajemnie. Często obiekty osadzone w kontrolerze same są kodem dostosowanym do własnych potrzeb za pomocą dziedziczenia.
8. Niewiele, ponieważ tak naprawdę był to pierwszy prototyp, ale metoda `lastName` musiałaby zostać zaktualizowana do nowego formatu nazw; konstruktor klasy `Person` musiałby zmienić domyślne zadanie na utworzenie pustej listy; a klasa `Manager` w swoim konstruktorze zamiast jednego ciągu znaków prawdopodobnie musiałaby przekazać listę zadań (oczywiście zmieniłby się również kod autotestu). Dobrą wiadomością jest to, że zmiany te musiałby zostać wprowadzone tylko w jednym miejscu — w naszych klasach, gdzie takie szczegóły są zawarte. Skrypty bazy danych powinny działać w takiej postaci, w jakiej są, ponieważ `shelves` obsługują dowolnie zagnieżdżone dane.
9. Klasy z niniejszego rozdziału można wykorzystać jako kod szablonu implementującego różne typy baz danych. Wystarczy zmienić ich cel, modyfikując konstruktory w taki sposób, by zapisywały one inne atrybuty, i udostępniając metody właściwe dla docelowego zastosowania. W przypadku bazy danych kontaktów możemy na przykład wykorzystać atrybuty takie, jak `name`, `address`, `birthday`, `phone` czy `email`, a także metody odpowiednie dla tego celu. Metoda o nazwie `sendmail` może na przykład wykorzystać moduł biblioteki standardowej Pythona `smtplib` do automatycznego wysłania po wywołaniu wiadomości e-mail do jednego z kontaktów (więcej informacji na temat takich narzędzi znajduje się w dokumentacji Pythona lub publikacjach poświęconych aplikacjom). Narzędzie `AttrDisplay` napisane w tym rozdziale można wykorzystać w jego aktualnej postaci do wyświetlenia obiektów, ponieważ celowo zostało ono zaprojektowane jako uniwersalne. Większość kodu baz danych `shelve` można wykorzystać do przechowywania obiektów, z niewielkimi tylko zmianami.

Rozdział 29. Szczegóły kodowania klas

Jeżeli nie zrozumiałeś jeszcze w pełni koncepcji programowania zorientowanego obiektowo w Pythonie, nie powinieneś się tym martwić — po krótkim wprowadzeniu teraz zaczniemy się bardziej szczegółowo zagłębiać w zagadnienia wprowadzone wcześniej. W tym oraz kolejnym rozdziale raz jeszcze przyjrzymy się mechanizmom związanym z klasami. W niniejszym rozdziale omówimy klasy, metody oraz dziedziczenie, rozszerzając niektóre najważniejsze zagadnienia wprowadzone w rozdziale 27. Ponieważ klasa jest naszym ostatnim narzędziem przestrzeni nazw, streścimy tutaj również koncepcję przestrzeni nazw i zasięgów Pythona.

W kolejnym rozdziale kontynuujemy dogłębne poznawanie mechanizmów klas, omawiając jeden ich szczególny aspekt — przeciążanie operatorów. Poza przedstawieniem szczegółów oba rozdziały — ten i kolejny — dają nam również okazję do zapoznania się z nieco większymi klasami niż omówione dotychczas.

Uwaga dotycząca treści: jeżeli czytasz tę książkę kolejno rozdział po rozdziale, zauważysz, że część tego rozdziału będzie przeglądem i streszczeniem tematów wprowadzonych w studium przypadku z poprzedniego rozdziału, do których wracamy tutaj, podając kilka nowych przykładów przeznaczonych głównie dla użytkowników stawiających pierwsze kroki w programowaniu zorientowanym obiektowo. Jeżeli masz w tej materii nieco większe doświadczenie, możesz mieć ochotę na pominięcie części tego rozdziału, ale koniecznie powinieneś zajrzeć przynajmniej do sekcji, w której omawiamy zasięgi przestrzeni nazw, ponieważ wyjaśniamy tam niektóre subtelności w modelu klas Pythona.

Instrukcja `class`

Choć instrukcja `class` w Pythonie może się z zewnątrz wydawać podobna do narzędzi z innych zorientowanych obiektowo języków programowania, przy bliższym jej poznaniu okazuje się inna od tego, do czego przyzwyczajeni są niektórzy programiści. Na przykład tak jak w C++ instrukcja `class` jest podstawowym narzędziem programowania zorientowanego obiektowo w Pythonie, jednak w przeciwieństwie do języka C++ `class` w Pythonie nie jest deklaracją. Tak jak `def`, instrukcja `class` tworzy obiekt, a także jest niejawnym przypisaniem — kiedy się ją wykonuje, generuje ona obiekt klasy i przechowuje referencję do niego w nazwie wykorzystanej w nagłówku. Podobnie jak `def`, instrukcja `class` jest prawdziwym kodem wykonywalnym — klasa nie istnieje, dopóki Python nie dotrze do definiującej ją instrukcji `class` i nie wykona jej (zazwyczaj kiedy importuje się moduł zawierający tę instrukcję, jednak nie wcześniej).

Ogólna forma

Instrukcja `class` jest instrukcją złożoną, z wciętym blokiem kodu, który zazwyczaj pojawia się pod nagłówkiem. W nagłówku po nazwie klasy w nawiasach wymienione są klasy nadzędne, rozzielone przecinkami. Podanie większej liczby klas nadzędnych prowadzi do dziedziczenia wielokrotnego (omówionego nieco bardziej formalnie w rozdziale 31.). Poniżej znajduje się ogólna postać instrukcji `class`.

```

class <nazwa>(klasa_nadrzędna, ...):
    # Przypisanie do nazwy
    attr = value
    # Współdzielone dane klasy
    def method(self, ...):
        # Metody
        self.attr = value
        # Dane instancji

```

Wewnątrz instrukcji `class` przypisania generują atrybuty klas, natomiast metody o specjalnych nazwach przeciążają operatory. Funkcja o nazwie `__init__` jest na przykład po zdefiniowaniu wywoływana w momencie konstruowania obiektu instancji.

Przykład

Jak widzieliśmy, klasy są w dużej mierze po prostu *przestrzeniami nazw* — to znaczy narzędziami służącymi do definiowania nazw (czyli atrybutów) eksportujących dane oraz logikę do klientów. Instrukcja `class` jednoznacznie definiuje przestrzeń nazw klasy. Tak jak w plikach modułów, instrukcje osadzone wewnątrz ciała instrukcji `class` tworzą jej atrybuty. Kiedy Python wykonuje instrukcję `class` (a nie wywołuje klasy), wykonuje wszystkie instrukcje w jej ciele, od góry do dołu. Przypisania mające miejsce w tym czasie tworzą w zasięgu lokalnym klasy zmienne, które stają się atrybutami w powiązanym obiekcie klasy. Z tego powodu klasy przypominają zarówno *moduły*, jak i *funkcje*.

- Tak jak funkcje, instrukcje `class` są zasięgami lokalnymi, w których istnieją zmienne utworzone przez zagnieżdżone operacje przypisania.
- Tak jak zmienne w module, zmienne przypisane w instrukcjach `class` stają się atrybutami obiektu klasy.

Podstawową cechą wyróżniającą klas jest to, że ich przestrzenie nazw są w Pythonie również podstawą *dziedziczenia*. Atrybuty referencji, które nie zostaną odnalezione w obiekcie klasy bądź instancji, pobierane są z innych klas.

Ponieważ `class` jest instrukcją złożoną, wewnątrz jej ciała można umieścić dowolną inną instrukcję — `print`, instrukcje przypisania, `if`, `def` i tak dalej. Wszystkie instrukcje w definicji klasy wykonywane są wtedy, gdy wykonywana jest instrukcja `class` (a nie wtedy, gdy klasa jest później wywoływana w celu utworzenia instancji). Przypisanie nazw wewnątrz instrukcji `class` tworzy atrybuty klasy, a zagnieżdżone instrukcje `def` tworzą metody klasy. Zasadniczo jednak dowolny typ przypisania nazwy na najwyższym poziomie instrukcji `class` tworzy atrybut o tej samej nazwie dla wynikowego obiektu klasy.

Przykładowo przypisania prostych obiektów niebędących funkcjami do atrybutów klasy tworzą *atrybuty danych*, współdzielone przez wszystkie instancje.

```

>>> class SharedData:
    spam = 42
    # Wygenerowanie atrybutu danych
    klasy

>>> x = SharedData()                      # Utworzenie dwóch instancji
>>> y = SharedData()
>>> x.spam, y.spam
spam (SharedData.spam)
(42, 42)

```

W kodzie powyżej, ponieważ zmienna `spam` przypisana jest na najwyższym poziomie instrukcji `class`, jest ona dodawana do klasy, dlatego będzie współdzielona przez wszystkie instancje.

Możemy ją zmodyfikować, przechodząc przez nazwę klasy, a także odnieść się do niej albo przez instancje, albo przez klasę^[1].

```
>>> SharedData.spam = 99  
>>> x.spam, y.spam, SharedData.spam  
(99, 99, 99)
```

Takie atrybuty klas można wykorzystać do zarządzania informacjami rozciągającymi się na wszystkie instancje — na przykład licznikiem liczby wygenerowanych instancji (pomyśl ten rozwiniemy w przykładzie z rozdziału 32.). Sprawdźmy teraz, co się stanie, jeżeli przypiszemy zmienną `spam` za pośrednictwem instancji, a nie klasy.

```
>>> x.spam = 88  
>>> x.spam, y.spam, SharedData.spam  
(88, 99, 99)
```

Przypisania do atrybutów instancji tworzą lub modyfikują zmienne w instancji, a nie we współdzielonej klasie. Mówiąc bardziej ogólnie, wyszukiwanie dziedziczenia występuje jedynie przy *referencjach* atrybutów, a nie przypisywaniu. Przypisanie atrybutu obiektu zawsze modyfikuje ten obiekt, a nie żaden inny^[2]. Przykładowo wyrażenie `y.spam` powoduje wyszukiwanie atrybutu w klasie za pomocą dziedziczenia, natomiast przypisanie wartości do `x.spam` dodaje zmienną do samego obiektu instancji `x`.

Poniżej znajduje się nieco szerszy przykład tego zachowania, który zachowuje tę samą zmienną w dwóch różnych miejscach. Założymy, że wykonamy poniższą instrukcję `class`.

```
class MixedNames:  
    # Zdefiniowanie klasy  
    data = 'mielonka'  
    # Przypisanie atrybutu klasy  
    def __init__(self, value):  
        # Przypisanie nazwy metody  
        self.data = value  
        # Przypisanie atrybutu instancji  
    def display(self):  
        print(self.data, MixedNames.data)      # Atrybut instancji, atrybut  
        # Klasy
```

Klasa ta zawiera dwie instrukcje `def` wiążące atrybuty klas z funkcjami metod. Zawiera również instrukcję przypisania `=`. Ponieważ operacja ta przypisuje zmienną `data` wewnętrz instrukcji `class`, znajduje się w lokalnym zasięgu klasy i staje się atrybutem obiektu klasy. Jak wszystkie atrybuty klasy, zmienna `data` jest dziedziczona i współdzielona przez wszystkie instancje klasy, które nie mają własnego atrybutu o nazwie `data`.

Kiedy tworzymy instancje tej klasy, atrybut `data` jest do nich dołączany przez przypisanie do `self.data` w metodzie konstruktora.

```
>>> x = MixedNames(1)                      # Utworzenie dwóch obiektów  
        # Instancji  
>>> y = MixedNames(2)                      # Każdy ma własne dane  
>>> x.display(); y.display()              # self.data jest inny,  
        # Subclass.data jest tym samym  
1 mielonka  
2 mielonka
```

W rezultacie zmienna `data` istnieje w dwóch miejscach — w obiektach instancji (utworzona przez przypisanie `self.data` w metodzie `__init__`), a także w klasie, po której jest dziedziczona przez instancje (utworzona przez przypisanie `data` w klasie). Metoda `display` klasy wyświetla obie wersje, najpierw kwalifikującinstancję `self`, a później klasę.

Wykorzystując te techniki do przechowania atrybutów w różnych obiektach, określamy ich zakres widoczności. Kiedy zmienne dołączane są do klas, są współdzielone. W instancjach zmienne zapisują dane dla tych instancji, a nie zachowanie współdzielone przez wszystkie instancje klasy. Choć wyszukiwanie dziedziczenia może dla nas wyszukiwać zmienne, zawsze możemy uzyskać dostęp do atrybutu z dowolnego miejsca drzewa poprzez bezpośrednie dotarcie do pożądanego obiektu.

W poprzednim przykładzie wyrażenia `x.data` lub `self.data` zwracają zmienną instancji, która normalnie ukrywa tę samą zmienną z klasy. Wyrażenie `MixedNames.data` pobiera natomiast zmienną klasy w sposób bezpośredni. W dalszej części rozdziału opiszymy jedno z najpopularniejszych zastosowań takiego wzorca kodu i bardziej szczegółowo objaśnimy, jak wdrożyliśmy go w poprzednim rozdziale.

Metody

Ponieważ znamy już funkcje, powinniśmy również znać metody klas. Metody to po prostu obiekty funkcji utworzone za pomocą instrukcji `def` zagnieżdzonych w ciele instrukcji `class`. Z abstrakcyjnego punktu widzenia metody udostępniają zachowanie, które dziedziczą instancje klasy. Z punktu widzenia programowania metody działają dokładnie tak samo jak proste funkcje, z jednym kluczowym wyjątkiem — pierwszy argument metody zawsze otrzymuje obiekt instancji, który jest domniemanym podmiotem wywołania metody.

Innymi słowy, Python automatycznie odwzorowuje wywołania metod instancji na funkcje metod klas w następujący sposób. Wywołania metod wykonywane za pośrednictwem instancji, takie jak poniższe:

```
instancia.metoda(argumenty...)
```

są automatycznie przekładane na wywołania funkcji metod klas w poniższej postaci:

```
klasa.metoda(instancia, argumenty...)
```

gdzie klasa ustalana jest na podstawie odnalezienia nazwy metody za pomocą procedury wyszukiwania dziedziczenia Pythona. Tak naprawdę w Pythonie poprawne są obie formy wywołania.

Poza normalnym sposobem dziedziczenia nazw atrybutów metod specjalny pierwszy argument jest jedną cechą wyróżniającą wywołań metod. W metodzie klasy pierwszy argument jest najczęściej, zgodnie z konwencją, nazywany `self` (z technicznego punktu widzenia znaczenie ma jedynie jego pozycja, nie nazwa). Argument ten udostępnia metodom punkt zaczepienia z powrotem do instancji będącej podmiotem wywołania. Ponieważ klasy generują wiele obiektów instancji, muszą stosować ten argument w celu zarządzania danymi różniącymi się między poszczególnymi instancjami.

Programiści języka C++ mogą rozpoznać argument Pythona `self` jako podobny do wskaźnika `this` z C++. W Pythonie `self` jest jednak zawsze obecne w kodzie w jawnym sposób — metody zawsze muszą przejść przez `self` w celu pobrania lub zmodyfikowania atrybutów instancji przetwarzanej w bieżącym wywołaniu metody. Taka jawną naturą `self` jest celowa — obecność tej nazwy sprawia, że oczywiste staje się, iż w skrypcie używamy nazw atrybutów instancji, a nie nazw z zakresu lokalnego czy globalnego.

Przykład metody

By wyjaśnić te koncepcje, przedstawmy je na przykładzie. Założymy, że definiujemy poniższą klasę.

```
class NextClass:  
    # Zdefiniowanie klasy  
    def printer(self, text):  
        # Zdefiniowanie metody  
        self.message = text  
        # Modyfikacja instancji  
        print(self.message)  
        # Dostęp do instancji
```

Nazwa `printer` odnosi się do obiektu funkcji. Ponieważ jest przypisana w zakresie instrukcji `class`, staje się atrybutem obiektu klasy i jest dziedziczona przez wszystkie instancje utworzone z tej klasy. Normalnie, ponieważ metody takie, jak `printer` zaprojektowane są do przetwarzania instancji, wywołujemy je przez instancje.

```
>>> x = NextClass()  
        # Utworzenie instancji  
>>> x.printer('wywołanie instancji')  
        # Wywołanie jej metody  
wywołanie instancji  
>>> x.message  
        # Modyfikacja instancji  
'wywołanie instancji'
```

Po wywołaniu metody za pomocą kwalifikacji instancji, jak w powyższym kodzie, metoda `printer` jest najpierw wyszukiwana za pomocą dziedziczenia, a później do jej argumentu `self` automatycznie zostaje przypisany obiekt instancji (`x`). Argument `text` otrzymuje łańcuch znaków przekazany w wywołaniu ('wywołanie instancji'). Ponieważ Python sam automatycznie przekazuje pierwszy argument do `self`, tak naprawdę musimy przekazać tylko jeden argument. Wewnątrz metody `printer` nazwa `self` wykorzystywana jest do dostępu lub ustawienia danych instancji, ponieważ odnosi się z powrotem do aktualnie przetwarzanej instancji.

Jak mogłeś się już przekonać, metody można wywoływać na dwa sposoby — za pośrednictwem instancji oraz za pośrednictwem samej klasy. Możemy na przykład również wywołać metodę `printer`, przechodząc nazwę klasy, pod warunkiem że w jawnym sposobie przekażemy instancję do argumentu `self`.

```
>>> NextClass.printer(x, 'wywołanie klasy') # Bezpośrednie wywołanie klasy  
wywołanie klasy  
>>> x.message  
        # Ponowna modyfikacja instancji  
'wywołanie klasy'
```

Wywołania przekazane przez instancję i klasę mają ten sam efekt, o ile w formie z klasą przekażemy ten sam obiekt instancji. Domyslnie otrzymamy komunikat o błędzie, jeśli spróbujemy wywołać metodę bez podania instancji.

```
>>> NextClass.printer('złe wywołanie')  
TypeError: unbound method printer() must be called with NextClass instance...
```

Wywoływanie konstruktorów klas nadzędnych

Metody są normalnie wywoływane za pośrednictwem instancji. Wywołania metod za pośrednictwem klas pojawiają się jednak w różnych specjalnych rolach. Jeden z często

spotykanych scenariuszy obejmuje metodę konstruktora. Metoda `__init__`, podobnie jak wszystkie atrybuty, wyszukiwana jest za pomocą dziedziczenia. Oznacza to, że w momencie konstrukcji Python lokalizuje i wywołuje tylko jedną metodę `__init__`. Jeśli konstruktory klas podrzędnych potrzebują gwarancji wykonania również logiki czasu konstrukcji z klasy nadrzędej, muszą wywołać metodę `__init__` klasy nadrzędej w sposób jawnym, za pośrednictwem tej klasy.

```
class Super:  
    def __init__(self, x):  
        ...kod domyślny...  
  
class Sub(Super): def __init__(self, x, y):  
    Super.__init__(self, x) # Wykonanie metody __init__  
    klasy nadrzędej  
    ...własny kod... # Wykonanie własnych działań  
    inicjalizacyjnych  
I = Sub(1, 2)
```

Jest to jeden z niewielu kontekstów, w których kod będzie prawdopodobnie wywoływał metodę przeciążania operatorów w sposób bezpośredni. Oczywiście powinniśmy wywoływać konstruktor klasy nadrzędej w ten sposób jedynie wtedy, gdy naprawdę *chcemy* ją wykonać — bez tego wywołania zostanie on całkowicie zastąpiony metodą klasy podrzędnej. Bardziej realistyczne zastosowanie tej techniki w praktyce znajduje się w przykładzie klasy Manager z poprzedniego rozdziału [3].

Inne możliwości wywoływania metod

Taki wzorzec wywoywania metod za pośrednictwem klas jest podstawą rozszerzania (a nie całkowitego zastępowania) odziedziczonego zachowania metody. Wymaga przekazania jawniej instancji, ponieważ wszystkie metody domyślnie tak robią. Technicznie rzecz biorąc, dzieje się tak, ponieważ metody są *metodami instancji*.



Zgodnie z tym, co przedstawiliśmy w ramce „A co z funkcją super?” w rozdziale 28., Python posiada również wbudowaną funkcję `super`, która pozwala na bardziej ogólne wywoływanie metod nadklasy, ale jej prezentację odłożymy aż do rozdziału 32. ze względu na jej złożoność i wiele wad. Więcej informacji znajdziesz we wspomnianej ramce; podstawowy sposób użycia tej funkcji jest nieodłącznie powiązany z wieloma powszechnie znymi kompromisami, choć funkcja posiada również bardzo zaawansowane i nieco ezoteryczne zastosowania, wymagające specjalnego, uniwersalnego sposobu jej wdrożenia. Z powodu tych problemów w naszej książce zamiast używać funkcji `super`, wolimy wywoływać klasy nadrzędne po jawnych nazwach; jeżeli dopiero zaczynasz korzystać z Pythona, polecamy takie samo podejście, zwłaszcza jeśli dopiero rozpoczynasz swoją przygodę z programowaniem zorientowanym obiektywnie. Jeżeli teraz dobrze opanujesz ten prosty, klasyczny sposób wywoływania klas nadrzędnich, łatwiej Ci będzie porównywać go później z innymi.

W rozdziale 32. spotkamy się również z nową opcją, dodaną w Pythonie 2.2 — *metodami statycznymi*, pozwalającym na tworzenie metod, które nie oczekują przekazywania obiektów instancji w pierwszym argumencie. Takie metody mogą działać jak proste funkcje bez instancji, ich nazwy są lokalne dla klas, w których są zapisane, i mogą one być wykorzystywane do zarządzania danymi klasy. Pokrewna koncepcja, którą spotkamy w tym samym rozdziale, czyli

metoda klasy, otrzymuje po wywołaniu klasę zamiast instancji, można ją wykorzystać do zarządzania danymi dla klasy i jest implikowana w metaklasach.

Są to jednak zarówno rozszerzenia zaawansowane, jak i zwykle opcjonalne. W normalnych warunkach instancja musi zawsze zostać przekazana do metody — czy to automatycznie, kiedy jest wywoływana przez instancję, czy ręcznie, gdy wywołujesz klasę.

Dziedziczenie

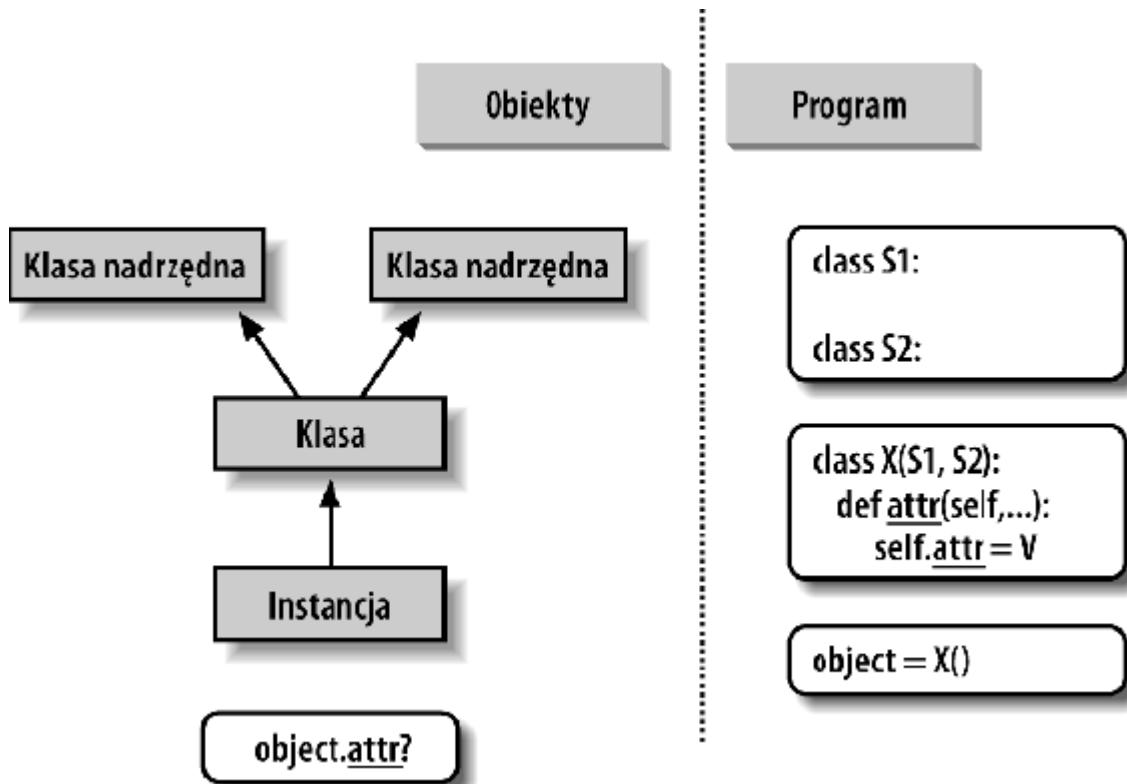
Oczywiście głównym celem istnienia przestrzeni nazw tworzonych przez instrukcję `class` jest obsługa dziedziczenia zmiennych. W tym podrozdziale rozszerzymy nieco informacje na temat niektórych mechanizmów oraz ról, jakie spełnia dziedziczenie atrybutów w Pythonie.

Jak mogłeś się już przekonać, w Pythonie dziedziczenie ma miejsce przy kwalifikacji obiektu i obejmuje przeszukiwanie drzewa definicji atrybutów (jednej lub większej liczby przestrzeni nazw). Za każdym razem gdy użyjemy wyrażenia w formie `obiekt.atrybut` (gdzie `obiekt` jest obiektem instancji lub klasy), Python przeszukuje przestrzeń nazw od dołu hierarchii do góry, rozpoczynając od obiektu i szukając pierwszego atrybutu o danej nazwie, jaki potrafi znaleźć. Obejmuje to również referencje do atrybutów `self` w metodach. Ponieważ definicje niżej w drzewie przesłaniają te umieszczone wyżej, dziedziczenie jest podstawą specjalizacji.

Tworzenie drzewa atrybutów

Na rysunku 29.1 przedstawiono sposób tworzenia drzew przestrzeni nazw oraz umieszczania w nich zmiennych. Mówiąc ogólnie:

- Atrybuty instancji generowane są przez przypisania do atrybutów `self` w metodach.
- Atrybuty klas tworzone są przez instrukcje (przypisania) w instrukcjach `class`.
- Łącza do klas nadrzędnych tworzy się, wymieniając te klasy w nawiasach w nagłówku instrukcji `class`.



Rysunek 29.1. Kod programu tworzy drzewo obiektów w pamięci, by móc je przeszukiwać pod kątem dziedziczenia atrybutów. Wywołanie klasy tworzy nową instancję pamiętającą swoją klasę. Wykonanie instrukcji `class` tworzy nową klasę, natomiast jej klasy nadziedzne wymienione są w nawiasach w nagłówku instrukcji `class`. Każda referencja do atrybutu wywołuje nowe wyszukiwanie od dołu drzewa do góry — nawet w przypadku atrybutów `self` wewnętrz metod klas

W rezultacie otrzymujemy drzewo przestrzeni nazw atrybutów prowadzące od instancji, przez klasę, która ją wygenerowała, do wszystkich klas nadziedzonych z nagłówka instrukcji `class`. Python szuka od dołu drzewa do góry, od instancji po klasy nadziedzne, za każdym razem, gdy użyjemy kwalifikacji w celu pobrania nazwy atrybutu z obiektu instancji [4].

Specjalizacja odziedziczonych metod

Opisany wyżej model dziedziczenia oparty na przeszukiwaniu drzew okazuje się doskonałym sposobem specjalizowania systemów. Ponieważ dziedziczenie odnajduje zmienne w klasach podrzędnych przed sprawdzeniem klas nadziedzonych, podklasy mogą zastępować zachowanie domyślne, redefiniując atrybuty swoich klas nadziedzonych. Tak naprawdę możemy budować całe systemy jako hierarchie klas rozszerzane przez dodawanie nowych zewnętrznych klas podrzędnych, zamiast wprowadzać modyfikacje istniejącej logiki w miejscu.

Pomysł redefiniowania odziedziczonych zmiennych prowadzi do różnych technik specjalizacyjnych. Klasa podrzędna może na przykład całkowicie zastępować odziedziczone atrybuty, udostępniać atrybuty, które klasa nadziedziona oczekuje znaleźć, a także rozszerzać metody klasy nadziedznej poprzez wywołania tej klasy z przesłoniętej metody. Widzieliśmy już, jak w praktyce działa zastępowanie. Poniżej znajduje się przykład rozszerzania.

```
>>> class Super:
```

```

def method(self):
    print('w Super.method')

>>> class Sub(Super):
    def method(self):                      # Przesłonienie metody
        print('początek Sub.method')       # Dodanie działań
        Super.method(self)                # Wykonanie działania domyślnego
        print('koniec Sub.method')

```

Kluczem są tu bezpośrednie wywołania metody klasy nadzędnej. Klasa Sub zastępuje metodę method z klasy Super za pomocą własnej, wyspecjalizowanej wersji. Wewnątrz tego zastąpienia klasa Sub wywołuje z powrotem wersję eksportowaną przez klasę nadzelną Super w celu wykonania zachowania domyślnego. Innymi słowy, Sub.method rozszerza jedynie zachowanie metody Super.method, zamiast całkowicie je zastępować.

```

>>> x = Super()                         # Utworzenie instancji klasy
Super

>>> x.method()                          # Wykonanie metody Super.method
w Super.method

>>> x = Sub()                           # Utworzenie instancji klasy Sub
                                         # Wykonuje Sub.method, co
wywołuje Super.method

początek Sub.method
w Super.method
koniec Sub.method

```

Taki wzorzec tworzenia rozszerzeń jest często wykorzystywany w przypadku konstruktorów. Przykład takiego działania znajduje się we wcześniejszym podrozdziale „Metody”.

Techniki interfejsów klas

Rozszerzanie jest jedną z metod wykonywania interfejsów klas nadzędnych. W poniższym pliku o nazwie *specialize.py* zdefiniowano większą liczbę klas w celu zilustrowania różnych popularnych technik.

Super

Definiuje funkcje `method` oraz `delegate`, które oczekują zmiennej `action` w klasach podrzędnych.

Inheritor

Nie udostępnia żadnych nowych zmiennych, dlatego wszystko ma zdefiniowane w klasie Super.

Replacer

Przesłania metodę `method` klasy Super za pomocą własnej wersji.

Extender

Dostosowuje metodę `method` klasy `Super` do własnych potrzeb, przesłaniając ją i wywołując w celu wykonania działania domyślnego.

Provider

Implementuje metodę `action` oczekiwana przez metodę `delegate` klasy `Super`.

Należy zapoznać się z każdą z tych klas, by zrozumieć, jak działają różne sposoby dostosowywania wspólnej klasy nadzędnej do własnych potrzeb. Poniżej widoczna jest zawartość pliku.

```
class Super:

    def method(self):
        print('w Super.method')                      # Zachowanie domyślne

    def delegate(self):
        self.action()                                # Oczekuje zdefiniowania

class Inheritor(Super):                         # Odziedziczenie wszystkich
    metod

    pass

class Replacer(Super):                          # Całkowite zastąpienie metody
    def method(self):
        print('w Replacer.method')

class Extender(Super):                         # Rozszerzenie działania metody
    def method(self):
        print('początek Extender.method')
        Super.method(self)
        print('koniec Extender.method')

class Provider(Super):                         # Uzupełnienie wymaganej metody
    def action(self):
        print('w Provider.action')

if __name__ == '__main__': for klass in (Inheritor, Replacer, Extender):
    print('\n' + klass.__name__ + '...')
    klass().method()
print('\nProvider...')

x = Provider()
x.delegate()
```

Warto zwrócić tutaj uwagę na dwie kwestie. Po pierwsze na to, w jaki sposób kod autotestu w końcowej części przykładu tworzy instancje trzech różnych klas w pętli `for`. Ponieważ klasy są obiektami, możemy je umieścić w krotce i tworzyć instancje w sposób uniwersalny bez konieczności używania żadnej dodatkowej składni (więcej na temat tej koncepcji później). Klasy mają również specjalny atrybut `__name__`, podobnie jak moduły. Jest on od razu ustawiany na łańcuch znaków zawierający nazwę z nagłówka instrukcji `class`. Oto co się stanie, kiedy wykonamy ten plik.

```
% python specialize.py
Inheritor...
  w Super.method
  Replacer...
    w Replacer.method
  Extender...
    początek Extender.method
    in Super.method
    koniec Extender.method
  Provider...
    w Provider.action
```

Abstrakcyjne klasy nadzędne

Dobre zrozumienie sposobu działania klasy `Provider` w poprzednim przykładzie może być tutaj kluczowe. Kiedy wywołamy metodę `delegate` za pośrednictwem instancji `Provider`, następują dwa niezależne wyszukiwania dziedziczenia.

1. W wywołaniu `x.delegate` Python odnajduje metodę `delegate` w klasie nadzędnej `Super`, szukając w instancji `Provider` i wyżej. Instancja `x` jest jak zwykle przekazywana do argumentu `self` metody.
2. Wewnątrz metody `Super.delegate` wywołanie `self.action` powoduje nowe, niezależne wyszukiwanie dziedziczenia w `self` i wyżej. Ponieważ `self` odnosi się do instancji `Provider`, metoda `action` zostaje odnaleziona w klasie podzędnej `Provider`.

Taka struktura kodu wypełniająca luki jest typowa dla programowania zorientowanego obiektowo. W bardziej realistycznych zastosowaniach wypełniona w ten sposób metoda może na przykład obsługiwać zdarzenia w graficznym interfejsie użytkownika, dostarczać dane do renderowania części strony internetowej, przetwarzać znaczniki w pliku XML itd. — podklasa zapewnia określone działania, a struktura obsługuje resztę ogólnych zadań.

Przynajmniej w kontekście metody `delegate` klasa nadzędna z tego przykładu jest czymś, co czasami nazywa się *abstrakcyjną klasą nadzędną* — klasą, która oczekuje, że część jej zachowania zostanie udostępniona przez klasę podzelną. Jeżeli oczekiwana metoda nie jest zdefiniowana w klasie podzędnej, Python zgłasza wyjątek niezdefiniowanej zmiennej, kiedy wyszukiwanie dziedziczenia zawiedzie.

Twórcy klas czasami czynią takie wymagania względem klas podzędnych bardziej oczywistymi za pomocą instrukcji `assert` lub zgłaszając wbudowany wyjątek `NotImplementedError` za pomocą instrukcji `raise` (instrukcje mogące zgłaszać wyjątki omówmy dogłębnie w kolejnej części książki). Poniżej zamieszczamy prosty przykład zastosowania instrukcji `assert`.

```
class Super:  
    def delegate(self):  
        self.action()  
    def action(self):
```

```

        assert False, 'działanie musi zostać zdefiniowane!'      # Jeżeli
wywołana jest ta wersja

>>> X = Super()
>>> X.delegate()

AssertionError: działanie musi zostać zdefiniowane!

```

Z instrukcją `assert` spotkamy się w rozdziałach 33. oraz 34. W skrócie, jeżeli pierwsze związane z nią wyrażenie jest fałszem, zgłasza ona wyjątek z podanym komunikatem o błędzie. Tutaj wyrażenie zawsze będzie fałszem, by wywołać komunikat o błędzie, jeśli metoda nie zostanie redefiniowana, a dziedziczenie zlokalizuje tę wersję. Alternatywnie niektóre klasy mogą w takich krótkich metodach po prostu bezpośrednio zgłosić wyjątek `NotImplemented` w celu zasygnalizowania błędu.

```

class Super:

    def delegate(self):
        self.action()

    def action(self):
        raise NotImplemented('działanie musi zostać zdefiniowane!')

>>> X = Super()
>>> X.delegate()

NotImplementedError: działanie musi zostać zdefiniowane!

```

W przypadku instancji klas podrzędnych nadal otrzymamy wyjątek, o ile klasa nadrzędna nie udostępnia oczekiwanej metody zastępującej metodę domyślną z klasy nadrzędnej.

```

>>> class Sub(Super): pass
>>> X = Sub()
>>> X.delegate()

NotImplementedError: działanie musi zostać zdefiniowane!

>>> class Sub(Super):

    def action(self): print('mielonka')

>>> X = Sub()
>>> X.delegate()

mielonka

```

Aby znaleźć nieco bardziej realistyczny przykład praktycznego zastosowania zagadnień omówionych w tym podrozdziale, powinieneś zajrzeć do hierarchii zwierząt z zoo w ćwiczeniu 8 na końcu rozdziału 32. oraz do jego rozwiązania w podrozdziale „Część VI. Klasy i programowanie zorientowane obiektywne” w dodatku D. Taksonomie takie jak powyższa są tradycyjnym sposobem wprowadzania podstaw programowania zorientowanego obiektywne, choć z oczywistych względów są raczej odległe od realnych zadań wykonywanych przez większość programistów (z całym szacunkiem dla wszystkich czytelników, którzy naprawdę pracują w zoo!).

Abstrakcyjne klasy nadrzędne z Pythona 3.x oraz 2.6+: wprowadzenie

W przypadku Pythona 2.6 oraz 3.0 opisane powyżej abstrakcyjne klasy nadrzędne (inaczej „abstrakcyjne klasy bazowe”), wymagające, by metody były uzupełniane w klasach podrzędnych, można implementować również za pomocą specjalnej składni klas. Sposób tworzenia ich kodu uzależniony jest w pewnym stopniu od wykorzystywanej wersji Pythona. W Pythonie 3.x wykorzystujemy argument ze słowem kluczowym w nagłówku instrukcji `class` w połączeniu ze specjalną składnią dekoratorów — obie techniki omówimy bardziej szczegółowo w kolejnej części książki.

```
from abc import ABCMeta, abstractmethod

class Super(metaclass=ABCMeta):

    @abstractmethod
    def method(self, ...):
        pass
```

W Pythonie 2.6 i 2.7 używamy zamiast tego atrybutu klasy.

```
class Super:

    __metaclass__ = ABCMeta

    @abstractmethod
    def method(self, ...):
        pass
```

Bez względu na sposób rezultat będzie ten sam — nie możemy utworzyć instancji, o ile metoda nie jest zdefiniowana niżej w drzewie klas. Poniżej znajduje się odpowiednik przykładu z poprzedniego podrozdziału, utworzony w Pythonie 3.x z wykorzystaniem składni specjalnej.

```
>>> from abc import ABCMeta, abstractmethod
>>>
>>> class Super(metaclass=ABCMeta):
        def delegate(self):
            self.action()
        @abstractmethod
        def action(self):
            pass
>>> X = Super()
TypeError: Can't instantiate abstract class Super with abstract methods
action
>>> class Sub(Super): pass
>>> X = Sub()
TypeError: Can't instantiate abstract class Sub with abstract methods action
>>> class Sub(Super):
        def action(self): print('mielonka')
>>> X = Sub()
```

```
>>> X.delegate()
```

mielonka

W przypadku zastosowania takiego kodu nie można utworzyć instancji, wywołując klasę z metodą abstrakcyjną, o ile wszystkie z metod abstrakcyjnych nie zostały zdefiniowane w klasach podanych. Choć rozwiązywanie to wymaga większej ilości kodu, jego zaletą jest to, że błędy wynikające z brakujących metod zwracane są, gdy próbujemy utworzyć instancję klasy, a nie później, gdy staramy się wywołać brakującą metodę. Opcję tę możemy także wykorzystać do zdefiniowania oczekiwanej interfejsu, automatycznie weryfikowanego w klasach klienta.

Niestety, rozwiązywanie to opiera się na dwóch zaawansowanych narzędziach języka, z którymi jeszcze się nie spotkaliśmy — *dekoratorach funkcji*, wprowadzonych w rozdziale 32. i omówionych dogłębnie w rozdziale 39., a także *deklaracjach metaklas*, o których wspominamy w rozdziale 32., a które przedstawimy w rozdziale 40. Pozostałe aspekty tej opcji odłożymy zatem na później. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona; warto także zapoznać się z gotowymi abstrakcyjnymi klasami nadzorowanymi udostępnianymi przez ten język.

Przestrzenie nazw — cała historia

Skoro już omówiliśmy obiekty klas oraz instancji, opowieść o przestrzeniach nazw w Pythonie jest pełna. Dla przypomnienia, szybko streszczę tutaj wszystkie reguły wykorzystywane w odnajdywaniu zmiennych. Pierwszą rzeczą, o jakiej należy pamiętać, jest to, że nazwy ze składnią kwalifikującą oraz bez niej są traktowane w inny sposób, a niektóre zakresy służą do inicjalizacji przestrzeni nazw obiektów.

- Nazwy bez zapisu kwalifikującego (na przykład `X`) związane są z zakresami.
- Nazwy z zapisem kwalifikującym (na przykład `obiekt.X`) wykorzystują przestrzeń nazw obiektów.
- Niektóre zakresy inicjalizują przestrzenie nazw obiektów (dla modułów oraz klas).

Wspomniane pojęcia czasami wchodzą w interakcje — na przykład w obiekcie `object.X`, który jest sprawdzany na podstawie zasięgów, a następnie `X` jest sprawdzany w obiektach wynikowych. Ponieważ zasięgi i przestrzenie nazw są niezbędne do zrozumienia kodu Pythona, podsumujemy te zasady bardziej szczegółowo.

Proste nazwy — globalne, o ile nie są przypisane

Jak już pokazywaliśmy, proste nazwy bez składni kwalifikującej zachowują się zgodnie z regułami zasięgów leksykalnych LEGB, które przedstawiliśmy przy okazji omawiania funkcji w rozdziale 17.

Przypisanie (X = wartość)

Domyślnie sprawia, że zmienna staje się lokalna. Tworzy lub modyfikuje zmienną `X` w bieżącym zasięgu lokalnym, o ile nie zostanie ona zadeklarowana jako globalna (lub typu `nonlocal` w wersji 3.x).

Referencja (X)

Wyszukuje zmienną `X` w bieżącym zasięgu lokalnym, następnie we wszystkich funkcjach zawierających, później w zasięgu globalnym, a na końcu — wbudowanym, zgodnie z regułą LEGB. Klassy zawierające nie są przeszukiwane, a zamiast tego nazwy klas są pobierane jako atrybuty obiektów.

Jak również wspominaliśmy w rozdziale 17., niektóre konstrukcje specjalne lokalizują nazwy jeszcze dalej (np. zmienne w niektórych elementach składanych i klauzulach `try`), ale zdecydowana większość nazw jest zgodna z regułą LEGB.

Nazwy atrybutów — przestrzenie nazw obiektów

Widzieliśmy także, że nazwy atrybutów ze składnią kwalifikującą odnoszą się do atrybutów określonych obiektów i stosują się do nich reguły dotyczące modułów oraz klas. W przypadku obiektów klas i instancji reguły związane z referencjami rozszerzane są w taki sposób, by obejmować procedurę wyszukiwania dziedziczenia.

Przypisanie (*obiekt.X = wartość*)

Tworzy lub modyfikuje atrybut o nazwie `X` w przestrzeni nazw *obiektu* kwalifikowanego i żadnego innego. Przechodzenie w górę drzewa dziedziczenia ma miejsce jedynie przy referencjach do atrybutu, a nie przypisaniach.

Referencja (*obiekt.X*)

W przypadku obiektów opartych na klasach szuka atrybutu o nazwie `X` w *obiekcie*, a następnie we wszystkich dostępnych klasach znajdujących się ponad nim, wykorzystując do tego procedurę wyszukiwania dziedziczenia. W przypadku obiektów niebędących klasami pobiera zmienną `X` bezpośrednio z *obiektu*.

Jak wspominaliśmy wcześniej, reguły opisane powyżej odnoszą się do normalnych, typowych sytuacji. Reguły atrybutów mogą się różnić w klasach, które wykorzystują bardziej zaawansowane narzędzia, szczególnie w klasach w nowym stylu, będących opcją w wersji 2.x i standardem w 3.x, które omówimy w rozdziale 32. Na przykład dziedziczenie referencji może być znacznie bogatsze, niż opisywaliśmy tutaj, gdy wdrażane są metaklasy, a klasy, które wykorzystują narzędzia do zarządzania atrybutami, takie jak właściwości, deskryptory czy metoda `__setattr__`, mogą przechwytywać i w dowolny sposób trasować przypisania atrybutów.

W rzeczywistości pewne elementy dziedziczenia są również wykorzystywane podczas przypisań do lokalizowania deskryptorów za pomocą metody `__set__` w klasach nowego stylu; takie narzędzia zastępują normalne zasady zarówno referencji, jak i przypisań. Narzędzia do zarządzania atrybutami dokładnie przeanalizujemy w rozdziale 38., a także sformalizujemy dziedziczenie i użycie deskryptorów w rozdziale 40. Na razie powinieneś się skupić na normalnych regułach podanych tutaj, które mają zastosowanie do zdecydowanej większości kodu w Pythonie, z jakim będziesz się na co dzień spotykał.

Zen przestrzeni nazw Pythona — przypisania klasyfikują zmienne

Mając osobne procedury wyszukiwania dla zmiennych ze składnią kwalifikującą oraz bez niej, a także wiele warstw do przeszukania w każdej z tych procedur, czasami trudno jest powiedzieć, gdzie trafi zmienna. W Pythonie miejsce *przypisania* zmiennej jest kluczowe — w pełni określa ono zakres lub obiekt, w którym zostanie umieszczone zmienna. Plik `manynames.py` ilustruje, jak zasada ta przekłada się na kod, a także streszcza zagadnienia dotyczące przestrzeni nazw, z którymi spotkaliśmy się w książce (pomijając przypadki specjalne, takie jak elementy składane):

```
# Plik manynames.py
X = 11                                         # Globalna zmienna/atribut z
modułu (X lub manynames.X)
def f():
    pass
```



```
#print(g.X)                                # PORAŻKA: widoczna tylko w
funkcji
```

Dane wyjściowe wyświetlane, kiedy plik jest wykonywany, zostały podane w komentarzach w kodzie. Warto je przejrzeć w celu sprawdzenia, do której zmiennej X uzyskujemy dostęp za każdym razem. Należy w szczególności sprawdzić, że możemy przejść klasę w celu uzyskania dostępu do jej atrybutu (C.X), jednak nigdy nie możemy pobrać zmiennych lokalnych w funkcjach lub metodach spoza ich instrukcji def. Zmienne lokalne są widoczne jedynie dla kodu wewnętrz tej samej instrukcji def i tak naprawdę istnieją w pamięci jedynie na czas wykonywania wywołania funkcji bądź metody.

Niektóre ze zmiennych zdefiniowanych przez ten plik są widoczne *poza nim* dla innych modułów, jednak należy pamiętać, że zanim uzyskamy dostęp do zmiennych z innego pliku, musimy ten plik najpierw zimportować — do tego w końcu służą moduły.

```
# Plik otherfile.py

import manynames

X = 66

print(X)                                     # 66: zmienna globalna tutaj

print(manyname.X)                            # 11: po zimportowaniu zmienne
globalne stają się atrybutami

manyname.f()                                  # 11: X z manyname, nie zmienna
globalna!

manyname.g()                                  # 22: zmienna lokalna z funkcji
innego pliku

print(manyname.C.X)                          # 33: atrybut klasy z innego modułu

I = manyname.C()

print(I.X)                                    # 33: nadal z klasy

I.m()

print(I.X)                                    # 55: teraz z instancji!
```

Warto zwrócić uwagę na to, w jaki sposób manyname.f() wyświetla zmienną X z modułu manyname, a nie X przypisaną w tym pliku. Zakresy są zawsze określane zgodnie z pozycją przypisania w pliku źródłowym (leksykalnie) i nigdy nie ma na nie wpływu to, co importuje co albo kto importuje kogo. Warto również odnotować, że własna zmienna X instancji nie jest tworzona, dopóki nie wywołamy I.m() — atrybuty, tak samo jak wszystkie zmienne, zaczynają istnieć przy przypisaniu, nie wcześniej. Normalnie tworzymy atrybuty instancji, przypisując je w metodzie konstruktora __init__ klasy, jednak nie jest to jedyna możliwość.

Wreszcie, jak wiemy z rozdziału 17., funkcja może modyfikować zmienne spoza siebie samej dzięki instrukcjom global i (w Pythonie 3.X) nonlocal. Instrukcje te umożliwiają dostęp do zapisu, jednak modyfikują również reguły dowiązań przestrzeni nazw przypisania.

```
X = 11                                      # Zmienna globalna w module

def g1():
    print(X)                                 # Referencja do zmiennej globalnej
    modułu (11)

def g2():
    global X
```

```

X = 22                                     # Modyfikacja zmiennej globalnej
modułu

def h1():
    X = 33                                 # Zmienna lokalna w funkcji

    def nested():
        print(X)                         # Referencja do zmiennej lokalnej w
        zakresie zawierającym (33)

    def h2():
        X = 33                             # Zmienna lokalna w funkcji

        def nested():
            nonlocal X                  # Instrukcja Pythona 3.x

            X = 44                      # Modyfikacja zmiennej lokalnej w
            zakresie zawierającym

```

Oczywiście nie powinniśmy na ogół używać tych samych nazw dla każdej zmiennej w naszym skrypcie. Ten przykład pokazuje jednak, że nawet jeśli tak zrobimy, przestrzenie nazw Pythona zadziałyają w taki sposób, by zapobiec konfliktom między zmiennymi przypisanyymi w różnych kontekstach.

Klasy zagnieżdżone — jeszcze kilka słów o regule LEGB

W poprzednim przykładzie podsumowaliśmy wpływ funkcji zagnieżdżonych na zasięgi, które badaliśmy w rozdziale 17. Okazuje się, że klasy można również zagnieżdzać — bardzo użyteczny wzorzec kodowania w niektórych typach programów, z implikacjami dotyczącymi zasięgów, które naturalnie wynikają z tego, co już wiesz, choć na pierwszy rzut oka może to nie być takie oczywiste. W tej sekcji spróbujemy to zilustrować na praktycznym przykładzie.

Choć klasy zwykle są kodowane na najwyższym poziomie modułu, czasami są również zagnieżdżane w funkcjach, które je generują, gdzie takie klasy są wykorzystywane do przechowywania stanów (jest to pewna wariacja na temat „funkcji fabrykujących”, które omawialiśmy w rozdziale 17.). W rozdziale 17. wspominaliśmy również, że instrukcje `class` wprowadzają nowe, lokalne zasięgi, podobnie jak instrukcje `def`, które podlegają tej samej zasadzie LEGB (przeszukiwanie zasięgów), co definicje funkcji.

Zasada ta dotyczy zarówno najwyższego poziomu samej klasy, jak i najwyższego poziomu metod w niej zagnieżdżonych. Oba elementy tworzą w tej regule warstwę *L* — są to normalne zasięgi lokalne, z dostępem do ich nazw, nazw w dowolnych funkcjach zawierających, nazw globalnych dla modułu zawierającego i nazw wbudowanych. Podobnie jak w przypadku modułów, lokalny zasięg klasy przekształca się w przestrzeń nazw atrybutów po uruchomieniu instrukcji `class`.

Chociaż klasy mają dostęp do zasięgów obejmujących funkcje, nie działają one jednak jako zasięgi zawierające dla kodu zagnieżdzonego w klasie: Python przeszukuje funkcje zawierające w poszukiwaniu przywoływanych nazw, ale *nigdy* nie przeszukuje klas zawierających. Oznacza to, że klasa jest zasięgiem lokalnym i ma dostęp do zawierających zasięgów lokalnych, ale nie spełnia roli zawierającego zasięgu lokalnego dla głębszej zagnieżdzonego kodu. Ponieważ wyszukiwanie nazw używanych w metodach pomija klasę zawierającą, atrybuty klas muszą być pobierane jako atrybuty obiektu za pomocą dziedziczenia.

Na przykład w poniżej funkcji `nester` wszystkie odwołania do `X` są kierowane do zasięgu globalnego, z wyjątkiem ostatniego, który pobiera redefinicję zasięgu lokalnego (kod tego

przykładu znajduje się w pliku *classscope.py*, a wyniki działania są opisane w dwóch ostatnich komentarzach):

```
X = 1

def nester():
    print(X)                                # Zmienna globalna: 1

    class C:
        print(X)                            # Nazwa globalna: 1
        def method1(self):
            print(X)                            # Nazwa globalna: 1
        def method2(self):
            X = 3                            # Przysłania zmienną globalną
            print(X)                            # Zmienna lokalna: 3

    I = C()
    I.method1()
    I.method2()

    print(X)                                # Zmienna globalna: 1
nester()                                    # Pozostałe: 1, 1, 1, 3
print('*'*40)
```

Zobaczmy jednak, co się stanie, gdy zmienimy przypisanie tej samej nazwy w zagnieżdżonych warstwach funkcji: redefinicje zmiennej X tworzą obiekty lokalne, które ukrywają te w otaczających zasięgach, tak jak w przypadku prostych funkcji zagnieżdżonych; zawierająca warstwa klasy nie zmienia tej reguły i w rzeczywistości nie ma dla niej znaczenia:

```
X = 1

def nester():
    X = 2                                # Przysłania zmienną globalną
    print(X)                            # Zmienna lokalna: 2

    class C:
        print(X)                            # W zawierającej def (nester): 2
        def method1(self):
            print(X)                            # W zawierającej def (nester): 2
        def method2(self):
            X = 3                            # Przysłania zawierającą (nester)
            print(X) # Local: 3

    I = C()
    I.method1()
    I.method2()
```

```

print(X)                                # Zmienna globalna: 1
nester()                                 # Pozostałe: 2, 2, 2, 3
print('*'*40)

A oto co się stanie, gdy zmienimy przypisanie tej samej nazwy w wielu miejscach po drodze:
przypisania w lokalnych zasięgach zarówno funkcji, jak i klas przesyłają nazwy globalne lub
nazwy lokalne funkcji zawierających, niezależnie od poziomu zagłędzienia:

X = 1

def nester():
    X = 2                                # Przesłania zmienną globalną
    print(X)                             # Zmienna lokalna: 2
    class C:
        X = 3                            # Klasa lokalna przesyłania nazwy
        funkcji nester: C.X lub I.X
        print(X)                         # Zmienna lokalna: 3
        def method1(self):
            print(X)                   # W zawierającym def (a nie 3 w
            klasie!): 2
            print(self.X)             # Dziedziczona zmienna lokalna
        klasy: 3
        def method2(self):
            X = 4                      # Przesłania zawierającą (funkcję
            nester, a nie klasę)
            print(X)                   # Zmienna lokalna: 4
            self.X = 5                 # Przesłania klasy
            print(self.X)             # Zlokalizowana w instancji: 5
    I = C()
    I.method1()
    I.method2()

    print(X)                                # Zmienna globalna: 1
    nester()                                 # Pozostałe: 2, 3, 2, 3, 4, 5
    print('*'*40)

```

Co najważniejsze, reguły wyszukiwania dla prostych nazw, takich jak `X`, nigdy nie powodują przeszukiwania zawierających instrukcji `class` — tylko `def`, moduły i nazwy wbudowane (to reguła LEGB, a nie CLEGB!). Na przykład w metodzie `method1` zmienna `X` znajduje się w instrukcji `def` poza klasą zawierającą, która ma taką samą nazwę w swoim zasięgu lokalnym. Aby dostać się do nazw przypisanych w klasie (np. metod), musimy pobrać je jako atrybuty obiektu klasy lub instancji, w tym przypadku za pomocą `self.X`.

Możesz wierzyć lub nie, ale zobaczysz przypadki użycia tego wzorca kodowania klas zagłędzonych w dalszej części tej książki, szczególnie w przypadku niektórych *dekoratorów* z

rozdziału 39. W tej roli funkcja zawierająca zwykle służy zarówno jako fabryka klas, jak i zapewnia przechowanie stanu do późniejszego użycia w zawieranej klasie lub jej metodach.

Słowniki przestrzeni nazw — przegląd

W rozdziale 23. wspomnieliśmy, że przestrzenie nazw są tak naprawdę zaimplementowane jako słowniki i udostępniane za pomocą wbudowanego atrybutu `__dict__`. W rozdziałach 27. i 28. mogłeś się przekonać, że tak samo jest w przypadku obiektów klas oraz instancji kwalifikacja atrybutów jest wewnętrznie operacją indeksowania słownika, a dziedziczenie atrybutów polega na przeszukaniu połączonych słowników. Tak naprawdę obiekty klas oraz instancji są w Pythonie w dużej mierze po prostu słownikami z łączami. Python udostępnia te słowniki wraz z łączami pomiędzy nimi, tak by można je było wykorzystywać w pewnych zaawansowanych zastosowaniach (na przykład do tworzenia narzędzi).

W poprzednim rozdziale wprowadziliśmy niektóre z tych narzędzi, ale aby w pełni zrozumieć, jak wewnętrznie działają atrybuty, przyjrzyjmy się sesji interaktywnej śledzącej rosnienie słowników przestrzeni nazw, kiedy w grę wchodzą klasy. Skoro jednak wiemy już więcej na temat metod i klas nadzędnych, trochę ją tutaj upiększymy. Najpierw zdefiniujemy klasę nadzijną oraz podzijną z metodami, które będą przechowywać dane w instancjach.

```
>>> class Super:
    def hello(self):
        self.data1 = 'mielonka'

>>> class Sub(Super):
    def hola(self):
        self.data2 = 'jajka'
```

Kiedy tworzymy instancję klasy podziennej, instancja ta ma na początku pusty słownik przestrzeni nazw, jednak zawiera łącza z powrotem do klasy, dzięki którym działa wyszukiwanie dziedziczenia. Tak naprawdę drzewo dziedziczenia jest w jawny sposób dostępne w atrybutach specjalnych, które można sprawdzać. Instancje mają atrybut `__class__` będący łączem do ich klasy, natomiast klasy mają atrybut `__bases__` będący krotką zawierającą łącza do znajdujących się wyżej w drzewie klas nadzędnych. Poniższy kod został wykonany w Pythonie 3.3; formaty nazw i pewne atrybuty wewnętrzne w Twojej wersji mogą się nieco różnić.

```
>>> X = Sub()
>>> X.__dict__                                     # Słownik przestrzeni nazw
instancji
{}

>>> X.__class__                                    # Klasa instancji
<class '__main__.Sub'>

>>> Sub.__bases__                                 # Klasa nadziedna klasy
(<class '__main__.super'>,)

>>> Super.__bases__                               # W Pythonie 2.x pusta krotka
()

(<class 'object'>,)
```

Kiedy klasy przypisują wartości do atrybutów `self`, zapełniają obiekty instancji. Atrybuty znajdują się zatem w słowniku przestrzeni nazw atrybutów instancji, a nie klas. Przestrzeń nazw obiektu instancji zapisuje dane różniące się z instancji na instancję, natomiast `self` jest punktem zaczepienia dla tej przestrzeni nazw.

```
>>> Y = Sub()
>>> X.hello()
>>> X.__dict__
{'data1': 'mielonka'}
>>> X.hola()
>>> X.__dict__
{'data2': 'jajka', 'data1': 'mielonka'}
>>> list(sub.__dict__.keys())
['__qualname__', '__module__', '__doc__', 'hola']
>>> list(Super.__dict__.keys())
['__module__', 'hello', '__dict__', '__qualname__', '__doc__', '__weakref__']
>>> Y.__dict__
{}
```

Warto zwrócić uwagę na specjalne nazwy z podwójnymi znakami `_` w słownikach klas. Python ustawia je automatycznie i w razie potrzeby możemy je odfiltrować za pomocą wyrażeń generatora, które widzieliśmy w rozdziałach 27. i 28. (nie będziemy ich tutaj powtarzać). Większość z nich nie jest wykorzystywana w typowych programach, jednak istnieją narzędzia używające niektórych z nich (na przykład `__doc__` przechowuje omówione w rozdziale 15. łańcuchy znaków dokumentacji).

Warto również zauważyć, że `Y` (druga instancja wykonana na początku tej serii) nadal ma na końcu pusty słownik przestrzeni nazw, pomimo że słownik instancji `X` został zapełniony przez przypisania w metodach. Jak wspomniano wcześniej, każda instancja ma niezależny słownik przestrzeni nazw, który na początku jest pusty i może z czasem zawierać zupełnie inne atrybuty od tych zapisanych przez słowniki przestrzeni nazw innych instancji tej samej klasy.

Ponieważ atrybuty są tak naprawdę wewnątrz Pythona kluczami słowników, istnieją dwa sposoby pobrania i przypisania ich wartości — za pomocą składni kwalifikującej lub indeksowania po kluczu.

```
>>> X.data1, X.__dict__['data1']
('mielonka', 'mielonka')
>>> X.data3 = 'tost'
>>> X.__dict__
{'data2': 'jajka', 'data3': 'tost', 'data1': 'mielonka' }
>>> X.__dict__['data3'] = 'szynka'
>>> X.data3
'szynka'
```

Jest tak jednak jedynie w przypadku atrybutów naprawdę dołączanych do *instancji*. Ponieważ składnia kwalifikująca pobierającą atrybuty wykonuje również wyszukiwanie *dziedziczenia*, ma dostęp do atrybutów, którego nie potrafi uzyskać indeksowanie słowników przestrzeni nazw. Przykładowo dostęp do odziedziczonego atrybutu `X.hello` za pomocą `X.__dict__['hello']` nie jest możliwy.

Spróbuj samodzielnie poeksperymentować z tymi specjalnymi atrybutami, aby lepiej poznać sposób, w jaki przestrzenie nazw faktycznie traktują atrybuty. Spróbuj także użyć tych obiektów z funkcją `dir`, którą poznaliśmy w poprzednich dwóch rozdziałach — wywołanie `dir(X)` jest podobne do `X.__dict__.keys()`, ale funkcja `dir` sortuje swoje wyniki i wyświetla niektóre odziedziczone i wbudowane atrybuty. Nawet jeżeli nigdy nie użyjesz ich w programach, które piszesz, to wiedza o tym, że są to zwykłe słowniki, może pomóc w poprawnym zdefiniowaniu przestrzeni nazw w ogóle.



W rozdziale 32. opowiadamy też o *slotach*, dosyć zaawansowanej funkcji klas w nowym stylu, która przechowuje atrybuty w instancjach, ale nie w słownikach przestrzeni nazw. Traktowanie ich jako atrybutów klasy jest bardzo kuszące i rzeczywiście pojawiają się one w przestrzeniach nazw klas, w których zarządzają wartościami poszczególnych instancji. Jak zobaczymy, sloty mogą uniemożliwić całkowite utworzenie słowników `__dict__` w instancji — jest to cecha, którą czasem muszą uwzględniać narzędzia ogólne używające narzędzi takich jak `dir` i `getattr`.

Łącza przestrzeni nazw — przechodzenie w górę drzewa klas

W poprzednim podrozdziale wprowadziliśmy atrybuty specjalne instancji oraz klas `__class__` i `__bases__` bez wyjaśnienia, dlaczego mogą one być dla nas istotne. Mówiąc w skrócie, atrybuty te pozwalały nam badać hierarchie dziedziczenia wewnętrz wlasnego kodu. Można je na przykład wykorzystać do wyświetlenia drzewa klas, jak w poniższym przykładzie dla Pythona 2.x i 3.x:

```
#!/usr/bin/python

"""

Plik classtree.py

Przechodzenie w górę drzewa dziedziczenia za pomocą łączy przestrzeni nazw,
wyświetla wyższe klasy nadzędne z odpowiednim wcięciem.

"""

def classtree(cls, indent):
    print('.' * indent, cls.__name__)                      # Wyświetlenie tu nazwy klasy
    for supercls in cls.__bases__:                          # Rekurencja po wszystkich
        klasach nadzędnych
            classtree(supercls, indent+3)                  # Może odwiedzić klasę nadzelną
        więcej niż raz
    def instancecltree(inst):
        print('Drzewo', inst)                            # Pokazanie instancji
        classtree(inst.__class__, 3)                     # Przejście do jej klasy
```

```

def selftest():

    class A: pass

    class B(A): pass

    class C(A): pass

    class D(B,C): pass

    class E: pass

    class F(D,E): pass

    instancemtree(B())
    instancemtree(F())

if __name__ == '__main__': selftest()

```

Funkcja `classtree` tego skryptu jest *rekurencyjna* — wyświetla nazwę klasy za pomocą atrybutu `__name__`, a następnie przechodzi do klasy nadzędnej, wywołując samą siebie. Pozwala to funkcji na przechodzenie drzew klas o dowolnym kształcie. Rekurencja pozwala przejść do góry i zatrzymuje się na głównych klasach nadzędnych, które mają puste atrybuty `__bases__`. Bez skorzystania z rekurencji każdy aktywny poziom funkcji otrzymuje własną kopię zasięgu lokalnego. Tutaj oznacza to, że `cls` oraz `indent` są inne na każdym poziomie `classtree`.

Większość tego pliku jest kodem samosprawdzającym. Kiedy wykona się go jako samodzielny plik w Pythonie 2.x, buduje on puste drzewo klas, wykonuje jego dwie instancje i wyświetla ich strukturę drzew klas.

```

C:\code> c:\python27\python classtree.py
Drzewo <__main__.B instance at 0x00000000022C3A88>
... B
..... A
Drzewo <__main__.F instance at 0x00000000022C3A88>
... F
..... D
..... B
..... A
..... C
..... A
..... E

```

Po wykonaniu w Pythonie 3.x drzewo zawiera domniemane klasy nadzędne `object`, które automatycznie dodawane są ponad klasami samodzielnymi, ponieważ w tej wersji wszystkie klasy są klasami w nowym stylu (więcej informacji na temat tej zmiany znajduje się w rozdziale 32.).

```

C:\code> c:\python33\python classtree.py
Drzewo <__main__.selftest.<locals>.B object at 0x00000000029216A0>
... B

```

```
.....A
.....object
Drzewo <__main__.selftest.<locals>.F object at 0x00000000029216A0>
...F
.....D
.....B
.....A
.....object
.....C
.....A
.....object
.....E
.....object
```

Wcięcia zaznaczone kropkami wykorzystywane są do oznaczenia wysokości drzewa klas. Moglibyśmy oczywiście poprawić nieco format danych wyjściowych i być może nawet naszkicować je w graficznym interfejsie użytkownika. Nawet w obecnej formie możemy zimportować te funkcje w dowolnym miejscu, w którym chcemy szybko wyświetlić drzewo klas.

```
C:\code> c:\python33\python
>>> class Emp: pass
...
>>> class Person(Emp): pass
>>> bob = Person()
>>> import classtree
>>> classtree.instancetree(bob)
Drzewo <__main__.Person object at 0x000000000298B6D8>
... Person
.... Emp
.....object
```

Bez względu na to, czy będziemy kiedyś korzystać z tych narzędzi, przykład ten demonstruje jeden z wielu sposobów wykorzystania atrybutów specjalnych udostępniających mechanizmy wewnętrzne interpretera. Kolejny zobaczymy, kiedy w pliku *lister.py* będziemy tworzyć uniwersalną klasę z narzędziami do wyświetlania — w podrozdziale „Dziedziczenie wielokrotne — klasy mieszane” w rozdziale 31. W przykładzie tym rozszerzymy tę technikę w taki sposób, by wyświetlała ona również atrybuty każdego obiektu drzewa klas i działała jako wspólna klasa nadrzędna.

W ostatniej części książki powrócimy do takich narzędzi w kontekście ogólnego budowania narzędzi Pythona w celu zaimplementowania prywatności atrybutów czy sprawdzania poprawności argumentów. Choć nie jest to przeznaczone dla każdego programisty Pythona,

dostęp do mechanizmów wewnętrznych daje narzędzia programistyczne o ogromnych możliwościach.

Raz jeszcze o notkach dokumentacyjnych

Ostatni przykład poprzedniego podrozdziału zawiera łańcuch znaków dokumentacji modułu, powinieneś jednak pamiętać, że notki dokumentacyjne można wykorzystywać także z przeznaczeniem dla komponentów klas. Notki dokumentacyjne, omówione szczegółowo w rozdziale 15., są literałami łańcuchów znaków, które pojawiają się na górze różnych struktur i są w Pythonie automatycznie zapisywane w atrybutach `__doc__` odpowiadających im obiektów. Działa to dla plików modułów, instrukcji `def` funkcji oraz klas i metod.

Skoro już wiemy coś więcej o klasach i metodach, plik `docstr.py` jest krótkim, choć wyczerpującym przykładem pokazującym miejsca, w których notki dokumentacyjne mogą się pojawić w kodzie. Mogą się one znaleźć w blokach z potrójnymi znakami cudzysłowów lub apostrofów albo w postaci prostszych literałów jednowierszowych:

```
"Jestem: docstr.__doc__"

def func(args):
    "Jestem: docstr.func.__doc__"
    pass

class spam:
    "Jestem: spam.__doc__ lub docstr.spam.__doc__ lub self.__doc__"
    def method(self):
        "Jestem: spam.method.__doc__ lub self.method.__doc__"
        print(self.__doc__)
        print(self.method.__doc__)
```

Podstawową zaletą notek dokumentacyjnych jest to, że są one dostępne w czasie działania programu. W ten sposób jeżeli coś zostało zapisane w kodzie jako notka dokumentacyjna, można w składni kwalifikującej zapisać obiekt oraz jego atrybut `__doc__` w celu pobrania jego dokumentacji.

```
>>> import docstr
>>> docstr.__doc__
'Jestem: docstr.__doc__'
>>> docstr.func.__doc__
'Jestem: docstr.func.__doc__'
>>> docstr.spam.__doc__
'Jestem: spam.__doc__ lub docstr.spam.__doc__ lub self.__doc__'
>>> docstr.spam.method.__doc__
'Jestem: spam.method.__doc__ lub self.method.__doc__'
>>> x = docstr.spam()
```

```
>>> x.method()  
Jestem: spam.__doc__ lub docstr.spam.__doc__ lub self.__doc__  
Jestem: spam.method.__doc__ lub self.method.__doc__
```

Omówienie narzędzia *PyDoc*, które potrafi formatować notki dokumentacyjne w raportach i w postaci stron internetowych, znajduje się w rozdziale 15. Poniżej zaprezentowano wyniki działania funkcji `help` dla naszego kodu w Pythonie 2.x. (Python 3.x pokazuje dodatkowe atrybuty odziedziczone po domniemanej klasie nadrzędnej `object` z modelu klas w nowym stylu. Można samodzielnie wykonać ten kod w Pythonie 3.x w celu zobaczenia dodatków z tej wersji; więcej informacji na temat tej różnicy znajduje się w rozdziale 32).

```
>>> help(docstr)  
Help on module docstr:  
  
NAME  
  
    docstr - Jestem: docstr.__doc__  
  
FILE  
  
    c:\code\docstr.py  
  
CLASSES  
  
    spam  
  
    class spam  
        | Jestem: spam.__doc__ lub docstr.spam.__doc__ lub self.__doc__  
        |  
        | Methods defined here:  
        |  
        | method(self)  
        | Jestem: spam.method.__doc__ lub self.method.__doc__  
  
FUNCTIONS  
  
    func(args)  
        Jestem: docstr.func.__doc__
```

Notki dokumentacyjne dostępne są w czasie działania, jednak są one — ze składniowego punktu widzenia — mniej elastyczne od komentarzy ze znakami `#` (które mogą się pojawić w dowolnym miejscu programu). Obie formy są przydatnymi narzędziami, a każda dokumentacja programu jest dobra (o ile, oczywiście, jest poprawna!). Jako ogólną regułę należy wykorzystywać notki dokumentacyjne na cele dokumentacji funkcjonalnej (wyjaśniającej, co robią obiekty), natomiast komentarze ze znakami `#` na cele dokumentacji poziomu mikro (mówiącej, jak działają niektóre bardziej złożone fragmenty kodu).

Klasy a moduły

Zakończymy ten rozdział, porównując krótko zagadnienia z dwóch ostatnich części niniejszej książki, poświęconych modułom oraz klasom. Ponieważ obie te kategorie dotyczą przestrzeni

nazw, rozróżnienie ich może być mylące. W skrócie:

- Moduły
 - implementują pakiety danych oraz logiki,
 - są tworzone przez pisanie plików Pythona lub rozszerzeń w innych językach,
 - można z nich korzystać po zaimportowaniu,
 - tworzą najwyższy poziom struktury programu w Pythonie.
- Klasa
 - implementują nowe, pełnowymiarowe obiekty,
 - są tworzone przez instrukcję `class`,
 - można z nich korzystać poprzez wywołanie,
 - zawsze istnieją wewnątrz modułu.

Klasy obsługują również dodatkowe opcje, których nie mają moduły, takie jak przeciążanie operatorów, generowanie wielu instancji oraz dziedziczenie. Choć zarówno klasy, jak i moduły są przestrzeniami nazw, powinniśmy już widzieć, że są zupełnie różnymi elementami. W kolejnych rozdziałach przekonasz się, jak różne mogą być klasy.

Podsumowanie rozdziału

Niniejszy rozdział zabrał nas na drugą, bardziej pogłębioną wycieczkę po mechanizmach programowania zorientowanego obiektowo języka Python. Dowiedzieliśmy się więcej o klasach, metodach oraz dziedziczeniu. Zakończyliśmy również omawianie przestrzeni nazw w Pythonie, rozszerzając je w taki sposób, by uwzględnić ich zastosowanie w klasach. Po drodze przyjrzaliśmy się pewnym bardziej zaawansowanym zagadnieniom, takim jak abstrakcyjne klasy nadrzędne, atrybuty danych klas, słowniki i łącza przestrzeni nazw oraz ręczne wywoływanie metod i konstruktorów klas nadrzędnych.

Skoro już wiemy tyle o mechanizmach tworzenia klas w kodzie Pythona, w rozdziale 30. przejdziemy do pewnego szczególnego aspektu tych mechanizmów — *przeciążania operatorów*. Potem omówimy często wykorzystywane wzorce projektowe — niektóre sposoby wykorzystywania klas oraz łączenia ich w celu zoptymalizowania ponownego użycia kodu. Przed przejściem dalej nie należy jednak zapominać o wykonaniu quizu podsumowującego rozdział.

Sprawdź swoją wiedzę — quiz

1. Co to jest abstrakcyjna klasa nadrzędna?
2. Co się dzieje, kiedy prosta instrukcja przypisania znajduje się na najwyższym poziomie instrukcji `class`?
3. Po co klasa miałaby ręcznie wywoływać metodę `__init__` z klasy nadrzędnej?
4. W jaki sposób można rozszerzać, zamiast całkowicie zastępować, odziedziczoną metodę?
5. Czym różni się zasięg lokalny klasy od zasięgu lokalnego funkcji?
6. Jak nazywała się stolica Asyrii?

Sprawdź swoją wiedzę – odpowiedzi

1. Abstrakcyjna klasa nadrzędna to klasa wywołująca metodę, jednak nie dziedzicząca ani nie definiującą jej — oczekuje ona, że metoda zostanie uzupełniona przez klasę podrzędną. Często wykorzystywana jest jako sposób uogólniania klas, kiedy jakieś zachowanie nie może zostać przewidziane aż do momentu zapisania w kodzie bardziej specyficznej klasy podrzędnnej. Platformy zorientowane obiektowo wykorzystują ten sposób do udostępniania operacji definiowanych przez klienta i możliwych do dostosowania do własnych potrzeb.
2. Kiedy prosta instrukcja przypisania (`X = Y`) pojawia się na najwyższym poziomie instrukcji `class`, dołącza ona atrybut danych do klasy (`Class.X`). Tak jak wszystkie atrybuty klas, będzie on współdzielony przez wszystkie instancje. Atrybuty danych nie są jednak wywoływalnymi funkcjami metod.
3. Klasa musi ręcznie wywoływać metodę `__init__` w klasie nadrzędnej, jeśli definiuje własny konstruktor `__init__`, ale nadal musi także uruchomić kod konstruktora klasy nadrzędnej. Python sam automatycznie wykonuje tylko *jeden* konstruktor — znajdujący się najbliżej w drzewie. Konstruktory klas nadrzędnych wywoływane są za pośrednictwem nazwy klasy, z ręcznym przekazaniem instancji `self`.
`Superclass.__init__(self, ...)`.
4. Aby rozszerzyć metodę odziedziczoną, zamiast całkowicie ją zastąpić, należy redefineować ją w klasie podrzędną i z tej nowej wersji wykonać ręczne wywołanie wersji metody z klasy nadrzędnej. Polega to na ręcznym przekazaniu instancji `self` do wersji metody z klasy nadrzędnej za pomocą kodu `Superclass.method(self, ...)`.
5. Klasa jest zasięgiem lokalnym i ma dostęp do zawierających zasięgów lokalnych, ale nie służy jako zawierający zasięg lokalny dla głębszej zagnieżdzonego kodu. Podobnie jak jest w przypadku modułów, zasięg lokalny klasy zmienia się w przestrzeń nazw atrybutów po wykonaniu instrukcji `class`.
6. Aszur (lub Kalaat Szirkat), Kalchu (lub Nimrud) lub przez krótki czas Dur-Szarrukin (lub Chorsabad), a na końcu Niniwa.

[1] Osoby, które używały w przeszłości języka C++, mogą rozpoznać to zachowanie jako podobne do „statycznych” składowych klas z tego języka — składowych przechowywanych w klasie i niezależnych od instancji. W Pythonie nie jest to nic specjalnego — wszystkie atrybuty klas są po prostu nazwami przypisanyimi w instrukcji `class`, bez względu na to, czy odnoszą się akurat do funkcji („metod” w C++), czy czegoś innego („składowych” w C++). W rozdziale 32. spotkamy się również z metodami statycznymi Pythona (podobnymi do tych z języka C++), które są po prostu funkcjami bez `self`, przetwarzającymi zazwyczaj atrybuty klas.

[2] Dopóki klasa nie zdefiniowała na nowo operacji przypisywania atrybutów, aby zrobić coś niestandardowego za pomocą metody przeciążającej operator `__setattr__` (omówionej w rozdziale 30.), lub nie używa zaawansowanych narzędzi atrybutów, takich jak właściwości i deskryptory (omówione w rozdziale 32. i 38.). W większej części tego rozdziału przedstawiamy normalny przypadek, który jest zupełnie wystarczający w tym punkcie książki, ale jak zobaczymy później, dzięki zastosowaniu tzw. punktów zaczepienia Python pozwala programom często odbiegać od normy.

[3] W tej samej klasie można utworzyć wiele metod `__init__`, jednak wykorzystana zostanie tylko ostatnia definicja. Więcej informacji na temat wielu definicji metod znajdziesz w rozdziale 31.

[4] Dwie uwagi: po pierwsze opis ten nie jest w stu procentach pełny, ponieważ możemy również tworzyć atrybuty instancji i klas, przypisując je do obiektów poza instrukcjami `class`. Jest to jednak rozwiązanie rzadziej spotykane i czasami bardziej podatne na błędy (zmiany nie są ograniczane do instrukcji `class`). W Pythonie wszystkie atrybuty są zawsze domyślnie dostępne. O prywatności zmiennych powiemy więcej w rozdziale 30., przy okazji omawiania metody `__setattr__`, w rozdziale 31., gdy spotkamy się ze zmiennymi `_X`, a także ponownie w rozdziale 39., gdzie zaimplementujemy ją za pomocą dekoratora klas.

Po drugie, jak wspominaliśmy w rozdziale 27., *pełne dziedziczenie* staje się bardziej skomplikowane, gdy dołożymy do tego bardziej zaawansowane zagadnienia, takie jak *metaklasy* i *deskryptory* — z tego powodu odłożymy formalną definicję do rozdziału 40. Jednak w najczęściej spotykanym scenariuszu jest to po prostu sposób przeddefiniowania, a tym samym dostosowania zachowania zakodowanego w klasach.

Rozdział 30. Przeciążanie operatorów

Niniejszy rozdział stanowi kontynuację przeglądu mechanizmów klas w Pythonie. Teraz będziemy zajmować się tematyką przeciążania operatorów. W poprzednich rozdziałach mieliśmy kilka przykładów przeciążania operatorów, w niniejszym rozdziale poznamy więcej szczegółów i przyjrzymy się przykładom powszechnych zastosowań tego mechanizmu. Nie będziemy opisywać szczegółowo wszystkich dostępnych metod, ale te, które omówimy, stanowią reprezentatywną próbę możliwości tego mechanizmu klas w Pythonie.

Podstawy

Przeciążanie operatorów to w rzeczywistości mechanizm *przechwytywania* wbudowanych metod klas: Python automatycznie wywołuje metody zdefiniowane przez użytkownika, gdy instancje klas występują w kontekście wbudowanych operatorów, a wartość zwracana z metody staje się następnie wynikiem operacji. Przeciążanie operatorów działa zgodnie z kilkoma podstawowymi koncepcjami:

- przeciążanie operatorów pozwala klasom przechwytywać operacje Pythona,
- klasy mogą przeciągać wszystkie operacje wyrażeń w Pythonie,
- klasy mogą również przeciągać wbudowane operacje, jak wyświetlanie znaków, wywołania funkcji, dostęp do atrybutów itp.,
- przeciążanie operatorów pozwala klasom definiowanym przez użytkownika działać w sposób zbliżony do typów wbudowanych,
- przeciążanie jest implementowane przez definiowanie metod klas o specjalnych nazwach.

Innymi słowy, gdy Python znajdzie w klasie metodę o nazwie odpowiedniej dla wykonywanego wyrażenia, wywoła ją i jej wynik zwróci jako wynik tego wyrażenia. Nowa klasa implementuje w utworzonym na jej podstawie obiekcie odpowiednie operacje.

Jak już wiemy, metody specjalne implementujące przeciążanie operatorów nigdy nie są wymagane w definicji klasy, nie ma też ich „domyślnych” implementacji (z wyjątkiem nielicznych, dziedziczonych przez niektóre klasy po klasie `object`): jeśli klasa nie ma zdefiniowanej takiej metody i nie dziedziczy jej po klasach nadrzędnych, oznacza to, że nie obsługuje danej operacji. W przypadku gdy jednak metody specjalne są zdefiniowane, pozwalają klasom emulować interfejsy obiektów wbudowanych, dzięki czemu implementacje mogą być bardziej spójne.

Konstruktory i wyrażenia – `__init__` i `__sub__`

Rozważmy następujący przykład: klasa `Number` zdefiniowana w pliku `number.py` udostępnia metodę przechwytyującą konstrukcję instancji (`__init__`) oraz drugą, implementującą wyrażenia odejmowania (`__sub__`). Te specjalne metody stanowią punkt zaczepienia pozwalający na użycie instancji klas w kontekście wbudowanych operatorów Pythona.

```

# Plik number.py

class Number:

    def __init__(self, start):                      # Wywoływana przy
Number(start)

        self.data = start

    def __sub__(self, other):                         # Wywoływana przy instancja -
inna

        return Number(self.data - other)             # Wynik jest nową instancją

>>> from number import Number
# Załadowanie klasy z modułu

>>> X = Number(5)
# Number.__init__(X, 5)

>>> Y = X - 2
# Number.__sub__(X, 2)

>>> Y.data
# Y jest nową instancją klasy
Number

3

```

Jak wspominałem w poprzednich rozdziałach, metoda `__init__` jest najczęstszym przykładem przeciążania operatorów w Pythonie, występuje w większości definiowanych klas i inicjuje tworzoną instancję, wykorzystując podane argumenty klasy. Metoda `__sub__` pełni rolę operatora binarnego, podobnie jak opisana w rozdziale 27. metoda `__add__`. Przechwytuje operacje odejmowania i zwraca nową instancję klasy (wcześniej wywoływaną jest metoda `__init__`).



Podczas tworzenia instancji najpierw jest wywoływana metoda `__new__`, która tworzy i zwraca obiekt instancji. Obiekt ten jest następnie przekazywany metodzie `__init__` do zainicjowania. Ponieważ metoda `__new__` posiada wbudowaną implementację i redefiniuje się ją tylko na potrzeby bardzo specyficznych zastosowań, niemal wszystkie klasy inicjuje się poprzez zdefiniowanie metody `__init__`. Jeden z przypadków użycia metody `__new__` poznamy w rozdziale 40. poświęconym *metaklasom*. Jest to dość rzadki przypadek wykorzystywany do dostosowywania procesu tworzenia instancji niemutowalnych typów danych.

Poznaliśmy już dość dokładnie metodę `__init__` i podstawowe operatory binarne, m.in. `__sub__`, dlatego nie będziemy ich tu ponownie przerabiać. W niniejszym rozdziale przyjrzymy się kilku innym dostępnym metodom oraz ich zastosowaniom w przykładach powszechnego użycia.

Często spotykane metody przeciążania operatorów

Prawie każde działanie, które możemy wykonać na obiektach wbudowanych, takich jak liczby całkowite czy listy, ma odpowiadającą mu metodę przeciążającą o specjalnej nazwie do zastosowania w klasach. W tabeli 30.1 zaprezentowano kilka najczęściej stosowanych metod, ale jest ich o wiele więcej. Tak naprawdę wiele z metod przeciążania operatorów ma kilka wersji (jak `__add__`, `__radd__` oraz `__iadd__` dla dodawania) i między innymi dlatego jest ich tak dużo. Wyczerpującą listę dostępnych specjalnych nazw metod można znaleźć w innych książkach poświęconych Pythonowi lub w dokumentacji języka.

Tabela 30.1. Często wykorzystywane metody przeciążania operatorów

Metoda	Przeciąża	Wywoływana dla
<code>__init__</code>	Konstruktor	Tworzenie obiektu — <code>X = Klasa(args)</code>
<code>__del__</code>	Destruktor	Zwolnienie obiektu X
<code>__add__</code>	Operator +	<code>X + Y, X += Y</code> , jeśli nie ma <code>__iadd__</code>
<code>__or__</code>	Operator (OR poziomu bitowego)	<code>X Y, X = Y</code> , jeśli nie ma <code>__ior__</code>
<code>__repr__, __str__</code>	Wyświetlanie, konwersje	<code>print X, repr(X), str(X)</code>
<code>__call__</code>	Wywołania funkcji	<code>X(*args, **kargs)</code>
<code>__getattr__</code>	Odczytanie atrybutu	<code>X.niezdefiniowany_atrybut</code>
<code>__setattr__</code>	Przypisanie atrybutu	<code>X.atrybut = wartość</code>
<code>__delattr__</code>	Usuwanie atrybutu	<code>del X.atrybut</code>
<code>__getattribute__</code>	Przechwytywanie atrybutu	<code>X.atrybut</code>
<code>__getitem__</code>	Indeksowanie, wycinanie, iteracje	<code>X[klucz], X[i:j]</code> , pętle for oraz inne iteracje, jeśli nie ma <code>__iter__</code>
<code>__setitem__</code>	Przypisanie indeksu i wycinka	<code>X[klucz] = wartość, X[i:j] = sekwencja</code>
<code>__delitem__</code>	Usuwanie indeksu i wycinka	<code>del X[klucz], del X[i:j]</code>
<code>__len__</code>	Długość	<code>len(X)</code> , testy prawdziwości, jeśli nie ma <code>__bool__</code>
<code>__bool__</code>	Testy logiczne	<code>bool(X)</code> , testy prawdziwości (odpowiednik <code>__nonzero__</code> w 2.x)
<code>__lt__, __gt__, __le__, __ge__, __eq__, __ne__</code>	Porównania	<code>X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y</code> (lub inaczej <code>__cmp__</code> — tylko w przypadku 2.x)
<code>__radd__</code>	Prawostronny operator +	<i>Nieinstancja + X</i>
<code>__iadd__</code>	Dodawanie w miejscu (rozszerzone)	<code>X += Y</code> (lub inaczej <code>__add__</code>)

<code>__iter__</code> , <code>__next__</code>	Konteksty iteracyjne	<code>I=iter(X), next(I); pętle for, jeśli nie ma __contains__, testy in, wszystkie listy składane, funkcje map (F,X), inne (<code>__next__</code> – odpowiednik <code>next</code> w 2.x)</code>
<code>__contains__</code>	Test przynależności	<code>item in X</code> (dowolny iterator)
<code>__index__</code>	Wartość całkowita	<code>hex(X), bin(X), oct(X), 0[X], 0[X:]</code> (zastępuje <code>__oct__</code> , <code>__hex__</code> itp. z Pythona 2.x)
<code>__enter__</code> , <code>__exit__</code>	Menedżer kontekstu (rozdział 34.)	<code>with obj as var:</code>
<code>__get__</code> , <code>__set__</code> , <code>__delete__</code>	Atrybuty deskryptorów (rozdział 38.)	<code>X.attr, X.attr = value, del X.attr</code>
<code>__new__</code>	Tworzenie instancji (rozdział 40.)	Tworzenie instancji, przed <code>__init__</code>

Wszystkie metody przeciążania operatorów mają nazwy rozpoczętymi się od dwóch znaków `_` i kończącymi się w ten sam sposób, co pozwala odróżnić je od innych nazw definiowanych w klasach. Odwzorowania ze specjalnych nazw metod na wyrażenia lub operacje są zdefiniowane w języku Python (i opisane w standardowej dokumentacji języka). Na przykład nazwa `__add__` zawsze odwzorowywana jest na wyrażenie z operatorem `+`, zgodnie z definicją języka, bez względu na to, co tak naprawdę robi kod metody `__add__`.



Choć wyrażenia wywołują metody operatorów, nie można zakładać, że usuwając element pośredni i wywołując te metody wprost, przyspieszy się działanie kodu. W rzeczywistości bezpośrednie wywołanie metody operatora może dwukrotnie spowolnić działanie programu z powodu obciążenia wprowadzanego przez wywołanie funkcji. Interpreter Pythona zapobiega takiemu efektowi i optymalizuje wykonywanie wbudowanych operacji.

Poniżej opisany jest przypadek metod `len` i `__len__`. Wykorzystany jest tu opisany w dodatku B program uruchamiający Pythona w systemie Windows oraz przedstawione w rozdziale 21. techniki pomiaru czasu działania kodu. W wersjach 3.x i 2.x bezpośrednie wywołanie metody `__len__` zajmuje dwa razy więcej czasu:

```
C:\code> py -3 -m timeit -n 1000 -r 5
          -s "L = list(range(100))" "x = L.__len__()"
1000 loops, best of 5: 0.134 usec per loop
C:\code> py -3 -m timeit -n 1000 -r 5
          -s "L = list(range(100))" "x = len(L)"
1000 loops, best of 5: 0.063 usec per loop
C:\code> py -2 -m timeit -n 1000 -r 5
          -s "L = list(range(100))" "x = L.__len__()"
1000 loops, best of 5: 0.117 usec per loop
```

```
C:\code> py -2 -m timeit -n 1000 -r 5
          -s "L = list(range(100))" "x = len(L)"
1000 loops, best of 5: 0.0596 usec per loop
```

Wbrew pozorom nie jest to dziwne. W pewnej znanej instytucji badawczej rzeczywiście spotkałem się z zaleceniem stosowania wolniejszej alternatywy na rzecz zwiększenia szybkości kodu!

Metody przeciążania operatorów mogą być dziedziczone z klas nadrzędnych, tak samo jak wszelkie inne metody. Wszystkie metody przeciążania operatorów są opcjonalne — jeśli którejkolwiek nie umieścimy w kodzie, operacja ta będzie po prostu nieobsługiwana w naszej klasie (i może powodować zwrócenie wyjątku, jeśli spróbujemy jej użyć). Niektóre powszechnie stosowane operacje, jak wyświetlanie, posiadają implementację domyślną, a dokładniej: są dziedziczone po klasie `object` (która jest dziedziczona przez wszystkie klasy w Pythonie 3.x), ale większość operatorów nie zadziała na instancjach klas, jeśli nie są w nich jawnie zdefiniowane odpowiednie metody przeciążające.

Większość metod przeciążania operatorów wykorzystywana jest jedynie w zaawansowanych programach wymagających, by obiekty zachowywały się jak te wbudowane, konstruktor `__init__` pojawia się jednak w większości klas. Przedstawmy teraz na przykładach kilka metod wymienionych w tabeli 30.1.

Indeksowanie i wycinanie — `__getitem__` i `__setitem__`

Pierwsza metoda, którą się zajmiemy, przypomina swoim działaniem sekwencję lub mapę. Jeśli w klasie jest zdefiniowana (lub dziedziczona po klasie nadrzędnej) metoda `__getitem__`, będzie ona automatycznie użyta w przypadkach prób wydobycia elementów po indeksach. Na przykład jeśli klasa `X` wystąpi w kontekście indeksowania `X[i]`, Python wywoła jej metodę `__getitem__`, przekazując `X` w pierwszym argumencie, a indeks w drugim.

Poniższa klasa zwraca kwadrat wartości indeksu (przykład jest dość nietypowy, ale dobrze ilustruje ogólny mechanizm):

```
>>> class Indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = Indexer()
>>> X[2]                                     # X[i] wywołuje X.__getitem__(i)
4
>>> for i in range(5):
...     print(X[i], end=' ')
__getitem__(X, i)
...
0 1 4 9 16
```

Wycinki

Co interesujące, metoda `__getitem__` jest wywoływana również w *wyrażeniach* wycinania (zawsze w wersji 3.x i warunkowo w 2.x, jeżeli nie zdefiniuje się bardziej konkretnych metod wycinających). Wbudowane typy obsługują wycinanie w ten sam sposób. Poniższy listing przedstawia operację wycinania przy wykorzystaniu dolnego i górnego zakresu oraz argumentu przesunięcia (więcej informacji na temat operacji wycinania można znaleźć w rozdziale 7.).

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                                     # Wycinanie z użyciem składni
wycinków
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
>>> L[::-1]
[5, 6, 7, 8]
>>> L[::2]
[5, 7, 9]
```

W rzeczywistości parametry wycinania są pakowane w specjalny *obiekt wycinka*, który jest przekazywany do instancji listy. Obiekt wycinka można przekazać ręcznie: składanie wycinków to jedynie składniowy skrót, ułatwiający pracę z obiektami wycinków.

```
>>> L[slice(2, 4)]                         # Wycinanie z użyciem obiektów
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

Ta obserwacja ma znaczenie dla klas implementujących metodę `__getitem__`, która w wersji 3.x jest wywoływana przy operacji indeksowania (wówczas otrzyma ona liczbę całkowitą) oraz wycinania (otrzyma obiekt wycinka). Nasz poprzedni przykład nie obsługuje wycinania, ponieważ metoda `__getitem__` zakłada, że otrzymała liczbę całkowitą, co naprawiamy w poniższym przykładzie. W przypadku indeksowania argument metody `__getitem__` jest liczbą całkowitą, jak poprzednio:

```
>>> class Indexer:
...     data = [5, 6, 7, 8, 9]
...     def __getitem__(self, index):    # Wywoływany przy indeksowaniu i
wycinaniu
...         print('getitem:', index)
...         return self.data[index]      # Realizuje dostęp po indeksie lub
wycinanie
```

```
...
>>> X = Indexer()
>>> X[0]                                     # Indeksowanie wysyła metodzie
__getitem__ liczbę całkowitą
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
```

W przypadku operacji wycinania nasza metoda otrzymuje obiekt wycinka, który po prostu przekazujemy do osadzonego obiektu `data` w nowym wyrażeniu indeksowania:

```
>>> X[2:4]                                     # Wycinanie wysyła metodzie
__getitem__ obiekt wycinka
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[::-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::-2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

W razie potrzeby metoda `__getitem__` może sprawdzać typ argumentu i wyodrębniać granice obiektu wycinka. Taki obiekt posiada atrybuty `start`, `stop` i `step`. Każdy z nich, jeżeli zostanie pominięty, przyjmuje wartość `None`.

```
>>> class Indexer:
    def __getitem__(self, index):
        if isinstance(index, int):           # Sprawdzenie trybu użycia
            print('indeks', index)
        else:
            print('wycinek', index.start, index.stop, index.step)
```

```
>>> X = Indexer()
>>> X[99]
indeks 99
>>> X[1:99:2]
wycinek 1 99 2
>>> X[1:]
wycinek 1 None None
```

Metoda `__setitem__` implementuje analogiczny mechanizm przypisywania wartości zarówno dla indeksów, jak i dla wycinków — w tym ostatnim przypadku w wersji 3.x (i zazwyczaj w wersji 2.x też) otrzymuje obiekt wycinka, który można wykorzystać bezpośrednio lub przekazać dalej tak samo do innego wyrażenia przypisującego wartość wycinkowi.

```
def __setitem__(self, index, value):      # Przechwytuje przypisania do indeksu
lub wycinka
...
    self.data[index] = value               # Przypisanie do indeksu lub wycinka
```

W rzeczywistości metoda `__getitem__` może być wywoływana automatycznie w operacjach bardziej zaawansowanych niż indeksowanie i wycinanie. Stanowi też opcję rezerwową dla *iteracji*, co omówimy w następnym podrozdziale. Wcześniej jednak przyjrzyjmy się ogólnie odmianie tych operacji w wersji 2.x i wyjaśnijmy pewien pozorny błąd.

Wycinanie i indeksowanie w Pythonie 2.x

W wersji 2.x klasy mogą również implementować również metody `__getslice__` i `__setslice__`, które odpowiadały za odczyt i przypisywanie wycinków. Metodom tym przekazywane były zakresy wycinków, a ich stosowanie było zalecane w kontekście wycinków zamiast używania metod `__getitem__` i `__setitem__`. Jednak w każdym innym przypadku kontekst jest taki sam jak w wersji 3.x. Na przykład, jeżeli nie ma metody `__getslice__` lub stosowana jest rozszerzona forma wycinka z trzema granicami, obiekt wycinka jest dalej tworzony i przekazywany metodzie `__getitem__`.

```
C:\code> c:\python27\python
>>> class Slicer:
    def __getitem__(self, index): print index
    def __getslice__(self, i, j): print i, j
    def __setslice__(self, i, j,seq): print i, j,seq
>>> Slicer()[1]           # Wywołuje __getitem__ z liczbą całkowitą, jak w
3.x
1
>>> Slicer()[1:9]         # Wywołuje __getslice__, jeżeli jest, lub
__getitem__ w przeciwnym wypadku
1 9
>>> Slicer()[1:9:2]       # Wywołuje __getitem__ z metodą slice, jak w 3.x
```

```
slice(1, 9, 2)
```

Te metody zostały *usunięte* w Pythonie 3.x i nawet w wersji 2.x zalecane jest stosowanie zamiast nich metod `__getitem__` i `__setitem__`, których argumenty są wykorzystywane zarówno do indeksowania, jak i wycinania. Kod jest wtedy kompatybilny w przód, jak również zapobiega się odmiennemu obsługiwaniu wycinków z dwoma lub trzema granicami. W większości klas nie trzeba stosować specjalnego kodu, ponieważ operacje indeksowania obsługują obiekty wycinków w nawiasach kwadratowych (jak w naszym przykładzie). Więcej przykładów przeciążania operacji wycinania można znaleźć w punkcie „Test przynależności — `__contains__`, `__iter__` i `__getitem__`”.

Metoda `__index__` w wersji 3.x nie służy do indeksowania!

Nie należy mylić metody `__index__` Pythona 3.x (której nazwa rzeczywiście wprowadza w błąd, a patrząc z perspektywy czasu, lepsza byłaby `__asindex__`) z przeciążaniem operacji indeksowania. Metoda ta służy bowiem do przekształcania obiektu na liczbę całkowitą i jest wykorzystywana przez funkcje przekształcające obiekty na ciągi liczb.

```
>>> class C:  
...     def __index__(self):  
...         return 255  
...  
>>> X = C()  
>>> hex(X)           # Wartość całkowita  
'0xff'  
>>> bin(X)  
'0b11111111'  
>>> oct(X)  
'0o377'
```

Metoda sprawdza nie jest stosowana do przechwytywania indeksowania instancji, np. za pomocą metody `__getitem__`, ale używa się jej w sytuacjach wymagających podania liczby całkowitej, między innymi przy indeksowaniu:

```
>>> ('C' * 256)[255]  
'C'  
>>> ('C' * 256)[X]      # Indeks po X (nie X[i])  
'C'  
>>> ('C' * 256)[X:]    # Indeks po X (nie X[i:])  
'C'
```

Ta metoda w 2.x działa tak samo, z tą różnicą, że nie jest wywoływana przez funkcje `hex` i `oct` (w 2.x należy w takim przypadku zdefiniować metody specjalne `__hex__` i `__oct__`).

Iteracja po indeksie — `__getitem__`

Istnieje sztuczka, która może być trudna do odgadnięcia dla początkujących, ale okazuje się niezwykle użyteczna. Jeżeli nie jest konkretnie zdefiniowana żadna z metod iteracyjnych opisanych w następnym rozdziale, instrukcja `for` przy każdej iteracji odczytuje jeden element, wykorzystując kolejny indeks, licząc od zera, aż zostanie wywołany wyjątek `IndexError` końca zakresu. Z tego powodu metoda `__getitem__` może być jednym ze sposobów implementacji iteratora w Pythonie: jeśli pętla `for` wykryje tę metodę, będzie ją wywoływać z kolejnymi indeksami.

Metoda `__getitem__` jest więc przykładem transakcji: „Kup jeden, drugi dostaniesz gratis” — każdy obiekt obsługujący indeksowanie potrafi również obsłużyć iterację:

```
>>> class StepperIndex:  
...     def __getitem__(self, i):  
...         return self.data[i]  
  
...  
>>> X = StepperIndex()                      # X jest instancją klasy StepperIndex  
>>> X.data = "Mielonka"  
>>>  
>>> X[1]                                     # Indeksowanie wywołuje __getitem__  
'p'  
>>> for item in X:                         # Pętle for wywołują __getitem__  
...     print(item, end=' ')                   # for indeksuje 0..N  
  
...  
M i e l o n k a
```

Tak naprawdę mamy tu przykład sytuacji: „Kup jeden, resztę dostaniesz gratis” — każda klasa obsługująca pętle `for` automatycznie obsługuje dowolne *konteksty iteracyjne* w Pythonie; wiele z nich poznaliśmy już we wcześniejszych rozdziałach (konteksty iteracyjne były prezentowane w rozdziale 14.). Na przykład instrukcja `in` sprawdzająca przynależność, lista składana, funkcja wbudowana `map`, przypisanie listy i krotki oraz w konstruktor typu automatycznie wywołują metodę `__getitem__`, jeśli jest zdefiniowana.

```
>>> 'p' in X                                # Wszystkie przykłady wywołują  
__getitem__  
True  
>>> [c for c in X]                          # Lista składana  
['S', 'p', 'a', 'm']  
>>> list(map(str.upper, X))                 # Funkcja (w 3.x niejawnie wywołuje  
list())  
['S', 'P', 'A', 'M']  
>>> (a, b, c, d) = X                         # Przypisania sekwencji  
>>> a, c, d
```

```

('S', 'a', 'm')
>>> list(X), tuple(X), ''.join(X)
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')
>>> X
<__main__.stepper object at 0x000000000297B630>

```

W praktyce ta technika może być użyta do tworzenia obiektów udostępniających interfejsy sekwencji oraz do definiowania dodatkowej logiki dla wbudowanych operatorów typów sekwencyjnych. Do tej koncepcji wróćmy przy okazji rozszerzania typów wbudowanych w rozdziale 32.

Obiekty iteratorów — `__iter__` i `__next__`

Mimo że metoda `__getitem__` z poprzedniego podrozdziału działa prawidłowo, jest jedynie mechanizmem „awaryjnym”, stosowanym przy iteracjach w przypadku braku lepszych metod. Wszystkie konteksty iteracyjne w nowszych wersjach Pythona w pierwszej kolejności próbują wywołać metodę `__iter__`, a jeśli jej nie znajdą, próbują `__getitem__`. Innymi słowy, protokół iteracyjny opisany w rozdziale 14. jest preferowany i ma przewagę nad kolejnym odczytywaniem elementów po indeksach. Jeśli obiekt nie obsługuje protokołu iteracyjnego, program próbuje odwołać się do indeksów. W większości przypadków również programista powinien preferować metodę `__iter__`: pozwala ona uogólnić konteksty iteracyjne w lepszy sposób, niż to jest możliwe z użyciem `__getitem__`.

Konteksty iteracyjne działają przez wywoływanie wbudowanej funkcji `iter`, wywołującą metodę `__iter__` obiektu iterowanego, która z kolei powinna zwracać iterator. Jeśli tak się stanie, Python w kolejnych iteracjach wywołuje metodę `__next__` iteratora zwracającą kolejne elementy, aż zostanie wywołany wyjątek `StopIteration`. Do obsługi iteratorów wykorzystywana jest funkcja wbudowana `next`, która może służyć do ręcznej iteracji w iteratorze. Wywołanie `next(I)` jest równoważne wywołaniu `I.__next__()`. Podstawowe informacje o tym modelu zostały pokazane na rysunku 14.1 w rozdziale 14.

Interfejs obiektu iteratora ma najwyższy priorytet i jest wykorzystywany w pierwszej kolejności. Dopiero gdy metoda `__iter__` nie zostanie odnaleziona w obiekcie, Python przełącza się w tryb wykorzystania metody `__getitem__`, w której kolejne elementy są pobierane przez pobieranie obiektów o kolejnych indeksach, aż zostanie wywołany wyjątek `IndexError`.



Uwaga na temat wersji: Jak wspominałem w rozdziale 14., w Pythonie 2.x metoda `I.__next__()` jest nazwana `I.next()`. Funkcja wbudowana `next(I)` została wprowadzona w celu zwiększenia przenośności kodu: w Pythonie 2.x wywołuje `I.next()`, natomiast w 3.x `I.__next__()`. W pozostałych szczegółach iteratory działają w 2.x tak samo jak w 3.x.

Iteratory zdefiniowane przez użytkownika

Po zdefiniowaniu mechanizmu `__iter__` klasy stają się iteratorami zdefiniowanymi przez użytkownika, implementując po prostu protokół iteracji przedstawiony w rozdziałach 14. oraz 20. (można w nich znaleźć więcej informacji na temat iteratorów). W poniższym pliku umieszczono na przykład klasę zdefiniowanego przez użytkownika iteratora, która generuje kolejne kwadraty liczb na żądanie, a nie wszystkie naraz. Zgodnie z wcześniejszą uwagą w

Pythonie 2.x należy definiować metodę `next`, a nie `__next__`, a w instrukcji `print` stosować jak zwykłe wiodący przecinek.

```
# Plik squares.py

class Squares:

    def __init__(self, start, stop):          # Zapisanie stanu przy
        utworzeniu

        self.value = start - 1

        self.stop = stop

    def __iter__(self):                      # Uzyskanie obiektu iteratora
        return self

    def next(self):                          # Zwrócenie kwadratu z każdą
        iteracją

        if self.value == self.stop:          # Również wywoływane przez
            funkcję wbudowaną next

            raise StopIteration

        self.value += 1

        return self.value ** 2
```

Po zimportowaniu klasy jej instancje są widoczne w kontekście iteracji, jak niżej:

```
C:\code> python
>>> from iters import Squares
>>> for i in Squares(1, 5):           # for wywołuje metodę iter(),
    która wywołuje __iter__()
...     print(i, end='')

...                                         # Każda iteracja wywołuje metodę
next()

1 4 9 16 25
```

W powyższym kodzie obiektem zwracanym przez metodę `__iter__` jest po prostu instancja `self`, ponieważ metoda `__next__` jest częścią tej klasy. W bardziej skomplikowanych scenariuszach obiekt iteratora może być zdefiniowany jako osobna klasa i obiekt z własnymi informacjami o stanie, co pozwala na obsługę wielu aktywnych iteracji po tych samych danych (przykład takiej sytuacji zobaczymy za moment). Koniec iteracji sygnalizowany jest za pomocą instrukcji `raise` Pythona (przedstawiona jest w rozdziale 29. i dokładnie opisana w następnej części książki). Ręczne iteracje działają tak samo dla obiektów iterowalnych zdefiniowanych przez użytkownika, jak i dla typów wbudowanych:

```
>>> X = Squares(1, 5)                 # Ręczna iteracja: tak działają pętle
>>> I = iter(X)                      # iter wywołuje __iter__
>>> next(I)                         # next wywołuje __next__ (w 3.x)

1
>>> next(I)
4
```

```

...pominięta część wyniku...

>>> next(I)
25
>>> next(I)                                # Można zastosować w instrukcji try:
StopIteration

```

Odpowiednik tego kodu, wykorzystujący metodę `__getitem__`, może być mniej naturalny, ponieważ pętla `for` wykonywałaby iterację po wszystkich wartościach przesunięcia od zera w górę. Przekazane wartości przesunięcia byłyby jedynie pośrednio związane z podanym przedziałem wartości ($0..N$ musiałoby zostać odwzorowane na `start..stop`). Ponieważ obiekty `__iter__` zachowują zarządzany w jawnym sposobie stan pomiędzy wywołaniami metody `next`, mogą być bardziej uniwersalne od metody `__getitem__`.

Z drugiej strony, iteratory oparte na metodzie `__iter__` mogą czasami być bardziej skomplikowane i mniej wygodne od `__getitem__`. Są właściwie zaprojektowane pod kątem iteracji, a nie dowolnego indeksowania — tak naprawdę wcale nie przeciążają one wyrażenia indeksującego. Można jednak zbierać generowane elementy w sekwencje, np. listy, i wykonywać na nich później inne operacje.

```

>>> X = Squares(1, 5)
>>> X[1]
AttributeError: Squares object does not support indexing
>>> list(X)[1]
4

```

Skanowanie pojedyncze i wielokrotne

Model z `__iter__` jest również implementacją wszystkich pozostałych kontekstów iteracyjnych, jakie widzieliśmy w przypadku metody `__getitem__` (testów przynależności, konstruktorów typów czy przypisania sekwencji). W przeciwieństwie do `__getitem__`, metoda `__iter__` zaprojektowana została do jednego przejścia, a nie wielu. Klasy w swoich kodach jawnie wybierają sposób skanowania.

Na przykład aktualna metoda `__iter__` klasy `Squares` zawsze zwraca atrybut `self` zawierający tylko jedną kopię stanu iteracji i dlatego wykonywana jest pojedyncza iteracja. Instancja klasy po zakończonej iteracji jest pusta. Ponownie wywołana metoda `__iter__` tej samej instancji znów zwraca atrybut `self` zawierający stan instancji, w jakim została pozostawiona. Dla każdej nowej iteracji zawsze trzeba tworzyć nowy iterowalny obiekt instancji.

```

>>> X = Squares(1, 5)                      # Utworzenie obiektu iteratora ze stanem
>>> [n for n in X]                        # Wyczerpuje elementy: metoda __iter__
zwraca self
[1, 4, 9, 16, 25]
>>> [n for n in X]                        # Teraz jest pusta: metoda __iter__ zwraca
self
[]
>>> [n for n in Squares(1, 5)]           # Utworzenie nowego obiektu iteratora
[1, 4, 9, 16, 25]

```

```
>>> list(Squares(1, 3))           # Nowy obiekt dla każdego nowego wywołania
__iter__
[1, 4, 9]
```

Aby bardziej bezpośrednio realizować wielokrotne iteracje, można w tym przykładzie zdefiniować dodatkową klasę lub zastosować inną technikę, co zrobimy za chwilę. Jednak po utworzeniu *nowej instancji* w każdej iteracji generowana jest nowa kopia stanu iteracji:

```
>>> 36 in Squares(1, 10)        # Inne konteksty iteracji
True
>>> a, b, c = Squares(1, 3)      # Za każdym razem wywoływana jest metoda
__iter__, a potem __next__
>>> a, b, c
(1, 4, 9)
>>> ':'.join(map(str, Squares(1, 5)))
'1:4:9:16:25'
```

Podobnie jak we wbudowanych operacjach, na przykład `map`, wykonujących pojedyncze skanowania, przekształcenie danych w listę skutkuje wykonaniem wielu skanowań, co wydłuża działanie kodu i zajmuje więcej pamięci (co może, choć nie musi, być istotne):

```
>>> X = Squares(1, 5)
>>> tuple(X), tuple(X)          # Iterator wyczerpany w drugiej krotce
((1, 4, 9, 16, 25), ())
>>> X = list(Squares(1, 5))
>>> tuple(X), tuple(X)
((1, 4, 9, 16, 25), (1, 4, 9, 16, 25))
```

Później, po porównaniu kilku rozwiązań, udoskonalimy ten kod, aby bardziej bezpośrednio wykorzystywał wielokrotne skanowanie.

Klasy i generatory

Warto zauważyć, że opisany przykład prawdopodobnie byłby prostszy, gdyby zapisano go z *funkcjami generatora* lub *wyrażeniami* (są to narzędzia przedstawione w rozdziale 20., automatycznie produkujące obiekty iteratorów i zachowujące lokalną zmienną stanu pomiędzy iteracjami):

```
>>> def gsquares(start, stop):
    for i in range(start, stop+1):
        yield i ** 2
>>> for i in gsquares(1, 5):
    print(i, end=' ')
1 4 9 16 25
>>> for i in (x ** 2 for x in range(1, 6)):
    print(i, end=' ')
```

1 4 9 16 25

W przeciwnieństwie do klasy, funkcje i wyrażenia generatora automatycznie zapisują stany pomiędzy iteracjami i tworzą metody wymagane przez protokół iteracyjny. W prostszych przypadkach, takich jak ten, w oczywisty sposób poprawia to zwięzłość kodu. Z drugiej strony, do bardziej skomplikowanych zastosowań lepiej nadają się klasy z jawnymi atrybutami i metodami, dodatkową strukturą, hierarchią dziedziczenia i możliwością wykonywania różnych działań.

Oczywiście w tym sztucznym przykładzie moglibyśmy tak naprawdę pominąć obie techniki i po prostu użyć pętli `for`, funkcji `map` czy listy składanej w celu zbudowania całej listy za jednym razem. Najlepsza i najszybsza metoda wykonania jakiegoś zadania w Pythonie jest często najłatwiejsza.

```
>>> [x ** 2 for x in range(1, 6)]  
[1, 4, 9, 16, 25]
```

Klasy mogą się jednak lepiej nadawać do modelowania bardziej skomplikowanych iteracji, w szczególności kiedy mogą one skorzystać z informacji o stanie oraz hierarchii dziedziczenia. Na przykład obiekt iteratora tworzący elementy w skomplikowanej bazie danych lub usłudze WWW może w większym stopniu wykorzystywać zalety klas. Poniżej omawiamy jeden taki przypadek użycia.

Wiele iteracji po jednym obiekcie

Wspomniałem wcześniej, że obiekt iteratora (z metodą `__next__`) można zdefiniować jako osobną klasę z własnymi informacjami o stanie, co pozwala na obsługę wielu aktywnych iteracji po tych samych danych. Warto rozważyć, co się stanie, kiedy przejdziemy typ wbudowany, taki jak łańcuch znaków.

```
>>> S = 'ace'  
>>> for x in S:  
...     for y in S:  
...         print x + y,  
  
aa ac ae ca cc ce ea ec ee
```

W powyższym kodzie zewnętrzna pętla `for` pobiera iterator z łańcucha znaków, wywołując `iter`, a każda pętla zagnieżdżona robi to samo w celu otrzymania niezależnego iteratora. Ponieważ każdy aktywny iterator ma swoje własne informacje o stanie, każda pętla może zachować swoją własną pozycję w łańcuchu znaków, niezależnie od pozostałych aktywnych pętli. Co więcej, nie trzeba za każdym razem tworzyć nowego ciągu ani przekształcać go w listę. Pojedynczy obiekt ciągu sam w sobie można wielokrotnie skanować.

Odpowiednie przykłady mieliśmy okazję poznać wcześniej, w rozdziałach 14. i 20. Na przykład funkcje i wyrażenia generatorów, jak również funkcje wbudowane `map` i `zip`, są w rzeczywistości obiektami generatorów, natomiast funkcja wbudowana `range` oraz typy wbudowane, jak listy, obsługują wielokrotne iteratory działające niezależnie od siebie, które mogą znajdować się w różnych pozycjach sekwencji.

Jeśli samodzielnie definiujemy własne iteratory w postaci klas, możemy zdecydować, czy chcemy obsługiwać pojedynczą iterację, czy wielokrotną. By uzyskać ten sam efekt za pomocą iteratorów zdefiniowanych przez użytkownika, `__iter__` musi po prostu zdefiniować nowy obiekt stanu dla iteratora, zamiast zwracać `self`.

Poniższy kod definiuje na przykład klasę iteratora `SkipObject`, która w czasie iteracji pomija co drugi element. Ponieważ obiekt iteratora tworzony jest od nowa dla każdej iteracji, obsługuje większą liczbę aktywnych pętli. (Kod zapisany jest w załączonym do książki pliku `skipper.py`).

```
#!/usr/bin/python3

# Plik skipper.py

class SkipObject:

    def __init__(self, wrapped):                      # Zapisanie elementu, który ma
                                                       # być użyty
        self.wrapped = wrapped

    def __iter__(self):
        return SkipIterator(self.wrapped)             # Za każdym razem nowy iterator

class SkipIterator:

    def __init__(self, wrapped):
        self.wrapped = wrapped                       # Informacje o stanie iteratora
        self.offset = 0

    def next(self):
        if self.offset >= len(self.wrapped):          # Zakończenie iteracji
            raise StopIteration
        else:
            item = self.wrapped[self.offset]           # Inaczej zwrócenie elementu i
                                                       # pominięcie
            self.offset += 2
            return item

    if __name__ == '__main__':
        alpha = 'abcdef'
        skipper = SkipObject(alpha)                  # Utworzenie obiektu pojemnika
        I = iter(skipper)                          # Utworzenie na nim iteratora
        print I.next(), I.next(), I.next()          # Odwiedzenie wartości
                                                       # przesunięcia 0, 2, 4
        for x in skipper:                         # for automatycznie wywołuje
                                                       # __iter__
            for y in skipper:                     # Zagnieżdżone for za każdym
                                                       # razem wywołują __iter__
                print x + y,                      # Każdy iterator ma własny stan
                                                       # i przesunięcie
```

Krótką uwagę dotyczącą przenośności: kod w przedstawionej wyżej postaci działa tylko w wersji 3.x. Aby był kompatybilny z wersją 2.x, należy zimportować do niego funkcję `print` z wersji 3.x, jak również użyć metody `next` zamiast `__next__` lub w obrębie klasy utworzyć alias

dla obu nazw. Kod będzie wtedy działał w obu wersjach (tak jak w przykładowym pliku `skipper_2x.py`).

```
#!python
from __future__ import print_function # Kompatybilność z wersjami 2.x i 3.x
...
class SkipIterator:
...
def __next__(self):
...
next = __next__ # Kompatybilność z wersjami 2.x i 3.x
```

Po wykonaniu przykład ten działa jak zagnieżdżone pętle z wbudowanymi łańcuchami znaków — każda aktywna pętla ma własną pozycję w łańcuchu znaków, ponieważ każda uzyskuje niezależny obiekt iteratora, zapisujący jej własne informacje o stanie.

```
C:\code> python skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

W przeciwnieństwie do tego rozwiązania nasz przykład z klasą `Squares` obsługuje tylko jedną aktywną iterację, o ile nie wywołamy tej klasy ponownie w zagnieżdżonych pętlach w celu otrzymania nowych obiektów. Tutaj jest tylko jeden obiekt klasy `SkipObject` z wieloma utworzonymi z niego obiektami iteratora.

Klasy i wycinki

Jak wcześniej, podobne rezultaty moglibyśmy uzyskać za pomocą narzędzi wbudowanych — na przykład wykorzystując wycinek z trzecim parametrem do pomijania elementów.

```
>>> S = 'abcdef'
>>> for x in S[::2]:
...     for y in S[::2]:                      # Nowe obiekty w każdej iteracji
...         print x + y,
...
aa ac ae ca cc ce ea ec ee
```

Nie jest to jednak to samo — z dwóch powodów. Po pierwsze, każde wyrażenie z wycinkiem *fizycznie przechowuje* listę wyników w całości w pamięci. Iteratory tworzą z kolei po jednej wartości na raz, co pozwala nam zaoszczędzić sporo miejsca w przypadku dużych list wyników. Po drugie, wycinki tworzą *nowe obiekty*, więc tak naprawdę wcale nie wykonujemy iteracji po tym samym obiekcie w wielu miejscach. By zbliżyć się do klasy, musielibyśmy uzyskać jeden obiekt do przechodzenia, wykonując wycinek z wyprzedzeniem.

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
```

```

>>> for x in S:
...     for y in S:                      # Ten sam obiekt, nowe iteratory
...         print x + y,
...
aa ac ae ca cc ce ea ec ee

```

Teraz bardziej przypomina to nasze rozwiązywanie oparte na klasie, jednak nadal musi ono przechować wynik wycinka w pamięci w całości (nie ma na razie żadnej formy generatora dla wycinków) i jest to tylko odpowiednik tego szczególnego przypadku pomijania co drugiego elementu.

Ponieważ iteratory mogą wykonać wszystko to, co klasy, są o wiele bardziej uniwersalne, niż mógłby to sugerować ten przykład. Bez względu na to, czy nasze aplikacje wymagają tego poziomu uniwersalności, iteratory definiowane przez użytkownika są narzędziem o dużych możliwościach — pozwalają na to, by dowolne obiekty wyglądały oraz zachowywały się jak inne spotkane w książce sekwencje oraz obiekty, po których można iterować. Moglibyśmy użyć tej techniki na przykład na obiekcie bazy danych, aby wykonać iteracje w celu pobierania danych z bazy, z wieloma kursorami w tym samym wyniku zapytania.

Alternatywa: metoda `_iter_` i instrukcja `yield`

A teraz coś dość tajemniczego, ale za to całkiem przydatnego. W niektórych aplikacjach można zminimalizować ilość kodu niezbędnego do utworzenia niestandardowej iteracji, łącząc opisaną tutaj metodę `_iter_` z instrukcją `yield` generatora, którą poznaliśmy w rozdziale 20. Funkcja generatora dobrze się do tego celu nadaje, ponieważ automatycznie zapisuje stan lokalnej zmiennej i tworzy wymagane metody iteratora, uzupełniając w ten sposób oferowane przez klasy narzędzia do zachowywania stanu.

Przypomnijmy sobie, że każda funkcja zawierająca instrukcję `yield` przekształca się w funkcję generatora. Nowa funkcja zwraca *obiekt generatora* automatycznie zachowującego lokalny zakres i pozycję kodu. Obiekt zawiera automatycznie utworzoną metodę `_iter_` zwracającą po prostu atrybut `self` oraz `_next_` (lub `next` w wersji 2.x), uruchamiającą funkcję i wznowiącą jej działanie od miejsca, w którym zostało przerwane:

```

>>> def gen(x):
    for i in range(x): yield i ** 2
>>> G = gen(5)                  # Utworzenie generatora z metodami __iter__ i
__next__
>>> G.__iter__() == G          # Obie metody są w tym samym obiekcie
True
>>> I = iter(G)                # Uruchomienie __iter__: generator zwraca samego
siebie
>>> next(I), next(I)          # Uruchomienie __next__ (next w 2.x)
(0, 1)
>>> list(gen(5))              # Kontekst iteracji automatycznie uruchamia iter
i next
[0, 1, 4, 9, 16]

```

Tak się dzieje nawet wtedy, gdy funkcja generatora z instrukcją `yield` jest metodą o nazwie `__iter__`. Taka metoda, wywołana za pomocą narzędzia iteracyjnego, zwraca nowy obiekt generatora zawierającego wymaganą metodę `__next__`. Dodatkowo funkcja generatora zakodowana jako metoda klasy ma dostęp do stanu zapisanego zarówno w atrybutach instancji, jak i zmiennych lokalnych.

Na przykład poniższa klasa jest równoważna początkowej iterowalnej klasie `Squares` zakodowanej w pliku `squares.py`.

```
# Plik squares_yield.py

class Squares:                      # Metoda __iter__ + instrukcja yield
    def __init__(self, start, stop):  # Automatyczna/domniemana metoda
        self.start = start
        self.stop = stop
    def __iter__(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

Aby uzyskać kompatybilność z wersją 2.x, nie trzeba tutaj tworzyć aliasu `next` dla metody `__next__`, ponieważ dzięki instrukcji `yield` metoda ta jest tworzona automatycznie lub jest domniemana. Jak poprzednio, pętle i inne narzędzia automatycznie iterują instancje tej klasy:

```
C:\code> python
>>> from squares_yield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')
1 4 9 16 25
```

Zawsze możemy zajrzeć w głąb kodu i sprawdzić, jak naprawdę działa on w kontekstach iteracyjnych. Jak zwykle uruchomienie instancji klasy za pomocą funkcji `iter` pozwala uzyskać wynik metody `__iter__`, ale w tym przypadku wynik ten jest obiektem generatora, zawierającym automatycznie utworzoną metodę `__next__` tego samego rodzaju co zawsze, gdy wywoływana jest funkcja generatora zawierająca instrukcję `yield`. Jedyna różnica polega na tym, że funkcja generatora jest automatycznie wywoływana przez funkcję `iter`. Wywołanie metody `next` obiektu powoduje utworzenie wyniku na żądanie:

```
>>> S = Squares(1, 5)      # Uruchomienie __init__: klasa zapisuje stan
instancji
>>> S
<squares_yield.Squares object at 0x000000000294B630>
>>> I = iter(S)           # Uruchomienie __iter__: zwrócenie generatora
>>> I
<generator object __iter__ at 0x00000000029A8CF0>
>>> next(I)
1
>>> next(I)               # Uruchomienie metody __next__ generatora
```

```

...itd...

>>> next(I)                      # Generator zawiera stan instancji i lokalnego
zakresu

StopIteration

```

Warto zauważyć, że metoda generatora może mieć inną nazwę niż `__iter__` i można ją uruchomić ręcznie np. za pomocą instrukcji `Squares(1,5).gen()`. Nazwa `__iter__` powoduje, że iteracja jest uruchamiana automatycznie przez narzędzie z pominięciem ręcznego pobrania atrybutu:

```

class Squares:                  # Odpowiednik o innej nazwie niż __iter__
(squares_manual.py)

    def __init__(...):
        ...

    def gen(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2

C:\code> python

>>> from squares_manual import Squares
>>> for i in Squares(1, 5).gen(): print(i, end=' ')
...te same wyniki...
>>> S = Squares(1, 5)
>>> I = iter(S.gen())           # Ręczne wywołanie generatora
>>> next(I)
...te same wyniki...

```

Zakodowanie generatora w postaci metody `__iter__` skutkuje usunięciem z kodu pośredniego elementu, jednak w obu przypadkach tworzony jest nowy obiekt generatora dla każdej iteracji:

- Jeżeli metoda `__iter__` istnieje, jest wywoływana podczas iteracji i zwraca nowy generator zawierający metodę `__next__`.
- Jeżeli metoda `__iter__` nie istnieje, kod tworzy generator, który zwraca samego siebie dla metody `__iter__`.

Jeżeli brzmi to skomplikowanie, można wrócić do rozdziału 20. zawierającego więcej informacji o instrukcji `yield` oraz generatorach i porównać je z opisaną wcześniej bardziej jawną wersją metody `__next__` z pliku `squares.py`. Należy zwrócić uwagę, że nowy plik `squares_yield.py` jest o cztery wiersze krótszy (ma siedem wierszy, a nie jedenaście) od poprzedniej wersji. W pewnym sensie w ten sposób można zmniejszać ilość niezbędnego kodu, podobnie jak za pomocą domknięć opisanych w rozdziale 17. Jednak w tym przypadku zamiast alternatywy dla klas wykorzystywane są techniki programowania funkcyjnego i obiektowego. Na przykład metoda generatora wciąż wykorzystuje atrybut `self`.

Dla niektórych użytkowników powyższy opis może kryć w sobie zbyt wiele magii. Wykorzystywany jest tu zarówno protokół iteracyjny, jak i tworzone są obiekty generatorów. Wbrew starym zasadom Pythona obie operacje są bardzo niejawne (patrz polecenie `import`

`this`). Niezależnie jednak od uznanych opinii warto znać iterowalne klasy, których nie dotyczy instrukcja `yield`, ponieważ są jawne, ogólne, a czasami mają większe zakresy.

Niemniej jednak technika wykorzystująca metodę `__iter__` i instrukcję `yield` jest skuteczna w szczególnych sytuacjach. Ma też pewną istotną zaletę, o której mowa w następnym podrozdziale.

Wielokrotne iteracje za pomocą instrukcji `yield`

Opisana w poprzednim podrozdziale niestandardowa klasa, iterowalna za pomocą metody `__iter__` i instrukcji `yield`, pozwala nie tylko pisać zwięzły kod, ale też ma ważną zaletę — automatycznie obsługuje wielokrotne, aktywne iteratory. Wynika to w naturalny sposób z faktu, że każde wywołanie metody `__iter__` jest wywołaniem funkcji generatora zwracającej nowy generator, który zawiera własną kopię lokalnego zakresu i stanu:

```
C:\code> python
>>> from squares_yield import Squares      # Wykorzystanie klasy Squares i
       __iter__/yield
>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I); next(I)
1
4
>>> J = iter(S)                          # Wielokrotne automatyczne
iteratory dzięki instrukcji yield
>>> next(J)
1
>>> next(I)                            # I jest niezależne od J: posiada
       lokalny stan
9
```

Choć funkcje generatora są iterowalnymi obiektami obsługującymi pojedyncze skanowanie, niejawne wywołania metody `__iter__` w kontekście iteracyjnym skutkują utworzeniem nowych generatorów obsługujących nowe, niezależne skanowania:

```
>>> S = Squares(1, 3)
>>> for i in S: # Each for calls __iter__
    for j in S:
        print('%s:%s' % (i, j), end=' ')
1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9
```

Aby osiągnąć ten sam efekt bez użycia instrukcji `yield`, należy utworzyć dodatkową klasę i zapisywać za jej pomocą stan iteratora jawnie i ręcznie przy użyciu technik opisanych w poprzednim podrozdziale (kod powiększy się wtedy do piętnastu wierszy, tj. będzie o osiem wierszy dłuższy od wersji wykorzystującej instrukcję `yield`):

```
# Plik squares_nonyield.py
class Squares:
```

```

    def __init__(self, start, stop):      # Generator bez instrukcji yield
        self.start = start                # Wielokrotne skanowanie: dodatkowy
obiekt
        self.stop = stop
    def __iter__(self):
        return SquaresIter(self.start, self.stop)
class SquaresIter:
    def __init__(self, start, stop):
        self.value = start - 1
        self.stop = stop
    def __next__(self):
        if self.value == self.stop:
            raise StopIteration
        self.value += 1
        return self.value ** 2

```

Poniższy kod działa tak samo jak jego poprzednia wersja z instrukcją `yield` i wielokrotnym skanowaniem, ale jest bardziej jawnym:

```

C:\code> python
>>> from squares_nonyield import Squares
>>> for i in Squares(1, 5): print(i, end=' ')
1 4 9 16 25
>>>
>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I); next(I)
1
4
>>> J = iter(S)                  # Wielokrotne iteratory bez instrukcji yield
>>> next(J)
1
>>> next(I)
9
>>> S = Squares(1, 3)
>>> for i in S:                 # Za każdym razem wywoływana jest metoda __iter__
    for j in S:

```

```

    print('%s:%s' % (i, j), end=' ')
1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9

```

Ponadto rozwiązywanie oparte na generatorach nie wymaga tworzenia dodatkowej klasy iteratora, jak w przykładzie `skipper.py`, dzięki automatycznym metodom i możliwości zachowywania stanu w lokalnych zmiennych (dodatkowo wymaga wpisania dziewięciu, a nie szesnastu wierszy kodu, jak w pierwotnej wersji):

```

# Plik skipper_yield.py

class SkipObject:                      # Kolejny generator oparty na __iter__ +
yield

def __init__(self, wrapped):           # Zakres instancji zachowywany normalnie
    self.wrapped = wrapped            # Lokalny stan zapisywany automatycznie

def __iter__(self):
    offset = 0
    while offset < len(self.wrapped):
        item = self.wrapped[offset]
        offset += 2
        yield item

```

Poniższy kod działa tak samo jak jego wersja z wielokrotnym skanowaniem bez instrukcji `yield`. Jest jednak krótszy i mniej jawnym:

```

C:\code> python
>>> from skipper_yield import SkipObject
>>> skipper = SkipObject('abcdef')
>>> I = iter(skipper)
>>> next(I); next(I); next(I)
'a'
'c'
'e'
>>> for x in skipper:          # Za każdym razem wywoływana jest metoda
    __iter__: nowy generator
        for y in skipper:
            print(x + y, end=' ')
aa ac ae ca cc ce ea ec ee

```

Oczywiście wszystkie powyższe przykłady są sztuczne i można je zastąpić prostszymi narzędziami, na przykład wyrażeniami listowymi, jednak utworzony w ten sposób kod może nie być tak skalowalny w bardziej praktycznych zastosowaniach. Warto przeanalizować i porównać alternatywne rozwiązania. Jak to często bywa w programowaniu, najlepsze narzędzie do danego zadania prawdopodobnie będzie najlepszym narzędziem dla zadania programisty!

Test przynależności — `__contains__`, `__iter__` i `__getitem__`

Mechanizm iteratorów ma jeszcze więcej możliwości. Przeciążanie operatorów często odbywa się *warstwowo* — klasy oferują szczegółowe metody lub bardziej uogólnione alternatywy wykorzystywane w zależności od dostępności. Na przykład:

- Porównania w 2.x wykorzystują metody `__lt__` do porównywania, czy inny obiekt jest mniejszy od instancji, ale bardziej ogólny interfejs dostarcza operator `__cmp__`. W Pythonie 3.x wykorzystywane są tylko metody szczegółowe, `__cmp__` nie jest już wywoływana (o czym w dalszej części rozdziału).
- Testy logiczne wykorzystują metodę `__bool__` (zwracającą wynik True/False), a jeśli instancja jej nie posiada, wywoływana jest metoda `__len__` (wynik niezerowy jest traktowany jako True). Jak się dowiemy w dalszej części rozdziału, Python 2.x działa tak samo, ale zamiast metody `__bool__` wykorzystuje metodę `__nonzero__`.

W przypadku iteratorów klasy implementują operator przynależności `in` i wykorzystują metodę `__iter__` albo `__getitem__`. W celu zastosowania bardziej zaawansowanej logiki klasy mogą również implementować metodę `__contains__`: gdy ta metoda jest dostępna, jest preferowana i ma przewagę nad `__iter__`, która z kolei ma pierwszeństwo przed `__getitem__`. Metoda `__contains__` powinna definiować przynależność na zasadzie kluczy słownika (wykorzystując szybkie wyszukiwanie) oraz jako mechanizm wyszukiwania w sekwencjach.

Przeanalizujmy poniższą klasę przystosowaną za pomocą wcześniejszych opisanych technik do wersji 2.x/3.x. Implementuje ona wszystkie trzy wspomniane metody i testuje przynależność obiektów oraz różne konteksty iteracyjne. Metody wyświetlają komunikaty diagnostyczne przy każdym wywołaniu.

```
# Plik contains.py
from __future__ import print_function          # Kompatybilność z wersją 2.x/3.x

class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i):                      # Metoda zastępcza do użycia
        przez iterację                            # oraz do indeksowania i
        wycinania                                # Metoda preferowana w iteracji
                                                # Pozwala na użycie tylko jednego
    def __iter__(self):                           # Metoda preferowana w iteracji
                                                # Pozwala na użycie tylko jednego
        iteratora                               self.ix = 0
                                                return self
    def __next__(self):
        print('next:', end='')
        if self.ix == len(self.data): raise StopIteration
```

```

        item = self.data[self.ix]
        self.ix += 1
        return item

    def __contains__(self, x):                      # Metoda preferowana w operacji
        'in'

        print('contains: ', end=' ')
        return x in self.data

    next = __next__                                # Kompatybilność z wersją 2.x/3.x

X = Iter([1, 2, 3, 4, 5])                      # Utworzenie instancji
print(3 in X)                                    # Przynależność
for i in X:                                      # Pętle for
    print(i, end=' | ')
print()
print([i ** 2 for i in X])                      # Inne konteksty iteracyjne
print(list(map(bin, X)))
I = iter(X)                                      # Ręczna iteracja (demonstracja
mechanizmu stosowanego                           # w kontekstach iteracyjnych)

while True:
    try:
        print(next(I), end=' @ ')
    except StopIteration:
        break

```

Obecnie klasa posiada metodę `__iter__` obsługującą wiele skanowań, ale w danej chwili aktywne może być tylko jedno (tj. zagnieżdżone pętle nie będą działać), ponieważ każda próba iteracji resetuje licznik skanowań. Teraz gdy znane jest działanie instrukcji `yield` z metodami iteracyjnymi, powinno być zrozumiałe, dlaczego poniższy kod, umożliwiający wykonywanie wielokrotnego skanowania, jest odpowiednikiem pokazanego wcześniej. Należy samodzielnie ocenić, czy warto rezygnować z jawności kodu na rzecz obsługi zagnieżdżonego skanowania i skrócenia kodu o sześć wierszy. (Kod znajduje się w pliku `contains_yield.py`).

```

class Iter:
    def __init__(self, value):
        self.data = value

    def __getitem__(self, i):                      # Opcja rezerwowa dla
iteracji,
        print('get[%s]: %i, end=%s')              # indeksów i wycinków
        return self.data[i]

    def __iter__(self):                            # Preferowane w iteracji

```

```

        print('iter=> next:', end='')           # Możliwe wielokrotne
aktywne iteratory

        for x in self.data:                   # Bez aliasu next dla
__next__

            yield x

            print('next:', end='')

    def __contains__(self, x):             # Preferowane w instrukcji
'in'

        print('contains: ', end='')

        return x in self.data

```

Powyższy skrypt uruchomiony w wersji 2.x lub 3.x wygeneruje wynik prezentowany na listingu poniżej. Metoda `__contains__` przechwytuje operację testu przynależności, metoda `__iter__` obsługuje konteksty iteracyjne, w których wywoływana jest metoda `__next__` (zakodowana jawnie lub niejawnie za pomocą instrukcji `yield`), natomiast metoda `__getitem__` nie jest nigdy wywoływana.

```

contains: True

iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:[0b1', '0b10', '0b11', '0b100',
'0b101']

iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:

```

Sprawdźmy, co się stanie w przypadku, gdy zostanie zakomentowana metoda `__contains__` — test przynależności jest realizowany przez metodę `__iter__`.

```

iter=> next:next:next:True

iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:[0b1', '0b10', '0b11', '0b100',
'0b101']

iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:

```

Na koniec przetestujmy wynik skryptu po ukryciu metod `__contains__` i `__iter__` — w takim przypadku wykorzystywana będzie metoda `__getitem__`, wywoływana z kolejnymi indeksami w kontekście testu przynależności oraz w iteracjach, aż do momentu zgłoszenia wyjątku `IndexError`:

```

get[0]:get[1]:get[2]:True

get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[0b1', '0b10', '0b11',
'0b100','0b101']

get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:

```

Jak widać, metoda `__getitem__` jest jeszcze bardziej ogólna: oprócz iteracji potrafi obsługiwać indeksowanie oraz tworzenie wycinków. Wyrażenia wycinające wywołują metodę `__getitem__` z obiektem wycinka określającym zakresy. Mechanizm działa tak samo dla typów wbudowanych, jak i dla klas zdefiniowanych przez użytkownika, zatem tworzenie wycinków zadziała w naszej klasie bez dodatkowego kodu:

```
>>> from contains import Iters
>>> X = Iters('spam')                      # Indeksowanie
>>> X[0]                                     # __getitem__(0)
get[0]:'s'
>>> 'spam'[1:]                                # Składnia wycinania
'pam'
>>> 'spam'[slice(1, None)]                  # Obiekt wycinka
'pam'
>>> X[1:]                                     # __getitem__(slice(..))
get[slice(1, None, None)]:'pam'
>>> X[::-1]
get[slice(None, -1, None)]:'spa'
>>> list(X)                                  # To też iteracja!
iter=> next:next:next:next:[‘s’, ‘p’, ‘a’, ‘m’]
```

W bardziej realistycznych sytuacjach związanych z iteratorami, które nie wykorzystują sekwencji, użycie metody `__iter__` może być łatwiejsze, ponieważ nie wymaga obsługi indeksu, natomiast metoda `__contains__` pozwala w niektórych przypadkach znacznie zoptymalizować testy przynależności.

Dostęp do atrybutów — `__getattr__` oraz `__setattr__`

W Pythonie klasy mogą również przechwytywać podstawowe odwołania do atrybutów, jeżeli zajdzie taka potrzeba. Po utworzeniu obiektu na podstawie klasy operacje odwołania, przypisania wartości i usunięcia atrybutu koduje się, wykorzystując wyrażenie z kropką: `obiekt.atrybut`. Prosty przykład widzieliśmy w rozdziale 28., natomiast tutaj rozwinemy szerzej ten temat.

Odwolania do atrybutów

Metoda `__getattr__` przechwytuje odwołania do atrybutów. Jest wywoływana z nazwą atrybutu jako łańcuchem znaków, zawsze gdy próbujemy zapisać w składni kwalifikującej instancję z *niezdefiniowaną* (nieistniejącą) nazwą atrybutu. *Nie* jest wywoływana, kiedy Python może odnaleźć atrybut, wykorzystując do tego procedurę wyszukiwania w drzewie dziedziczenia.

Ze względu na to zachowanie metoda `__getattr__` przydaje się jako punkt zaczepienia dla odpowiadania na żądania atrybutów w sposób ogólny. Jest powszechnie stosowana do

delegowania wywołań z pośredniczącego obiektu kontrolera (opisanego w rozdziale 28. we wprowadzeniu do *delegacji*) do osadzonych (lub „opakowanych”) obiektów. Metodę tę można również wykorzystywać do przystosowywania klasy do interfejsu lub do *dodawania* metod dostępowych do atrybutów, tj. kodu weryfikującego lub wyliczającego wartości atrybutów po użyciu ich w instrukcji wykorzystującej prosty zapis z kropką.

Mechanizm realizujący powyższe cele jest prosty. Poniższa klasa przechwytuje odwołania do atrybutów i wylicza dynamicznie ich wartości. Jeżeli atrybut nie jest obsługiwany, klasa zgłasza błąd za pomocą instrukcji `raise` użytej wcześniej w tym rozdziale w części poświęconej iteratorm (dokładnie opisanej w części VII książki).

```
>>> class Empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 40
...         else:
...             raise AttributeError, attrname
...
>>> X = Empty( )
>>> X.age
40
>>> X.name
...pominieto tekst błędu...
AttributeError: name
```

W powyższym kodzie klasa `Empty` oraz jej instancja `X` nie mają prawdziwych własnych atrybutów, dlatego dostęp do `X.age` powoduje przekazanie do metody `__getattr__`. Do atrybutu `self` zostaje przypisana instancja (`X`), a do atrybutu `attrname` przypisuje się łańcuch znaków nazwy niezdefiniowanego atrybutu ("age"). Klasa sprawia, że `age` wygląda jak prawdziwy atrybut, zwracając prawdziwą wartość (40) jako wynik wyrażenia ze składnią kwalifikującą `X.age`. W rezultacie `age` staje się *dynamicznie obliczonym* atrybutem, tj. którego wartość jest wyliczana za pomocą kodu.

W przypadku atrybutów, z którymi klasa nie wie, jak sobie radzić, metoda `__getattr__` zgłasza wbudowany wyjątek `AttributeError`, który przekazuje Pythonowi, że są to faktycznie zmienne niezdefiniowane — próba uzyskania `X.name` powoduje wystąpienie błędu. Z metodą `__getattr__` spotkamy się ponownie, kiedy zobaczymy działanie delegacji oraz właściwości w kolejnych dwóch rozdziałach. Teraz zajmiemy się odpowiednimi narzędziami.

Przypisywanie wartości i usuwanie atrybutów

Podobna metoda przeciążania operatorów o nazwie `__setattr__` przechwytuje *wszystkie* przypisania atrybutów. Jeśli jest ona zdefiniowana lub odziedziczona, `self.atrybut = wartość` staje się `self.__setattr__('atrybut', wartość)`. Metoda ta, podobnie jak `__getattr__`, umożliwia przechwytywanie zmian wprowadzanych w atrybutach i w razie potrzeby weryfikowanie i przekształcanie ich wartości.

Metoda jest nieco trudniejsza w użyciu, ponieważ przypisanie do dowolnych atrybutów `self` wewnętrz wywołania metody `__setattr__` ponownie wywołuje `__setattr__`, powodując nieskończoną *petlę rekurencji* (i w końcu wyjątek przepelenienia stosu). W rzeczywistości

dotyczy to wszystkich operacji przypisania wartości atrybutowi `self` w dowolnym miejscu klasy. Operacje te są kierowane do metody `__setattr__`, nawet jeżeli są wykonywane w innych metodach lub wywołują metodę `__setattr__` w pierwszej kolejności. Należy pamiętać, że przechwytywane są operacje przypisania *wszystkich* atrybutów.

Jeśli chcemy korzystać z tej metody, należy zapobiec powstaniu pętli poprzez zakodowanie przypisania atrybutu instancji w postaci przypisania klucza słownika. Oznacza to, że trzeba użyć zapisu `self.__dict__['name'] = x` zamiast `self.name = x`. Ponieważ wartość nie jest przypisywana samemu atrybutowi `__dict__`, zapobiega się w ten sposób powstawaniu pętli:

```
>>> class Accesscontrol:
...     def __setattr__(self, attr, value):
...         if attr == 'age':
...             self.__dict__[attr] = value + 10    # Niedozwolone self.name=values
...         else:
...             raise AttributeError(attr + ' nie jest dozwolony')
...
>>> X = Accesscontrol( )
>>> X.age = 40                                # Wywołuje __setattr__
>>> X.age
50
>>> X.name = 'amadeusz'
...tekst pominięto...
AttributeError: name nie jest dozwolony
```

Jeżeli wiersz przypisujący wartość do atrybutu `__dict__` zostanie zmieniony na jeden z poniższych, powstanie nieskończona pętla rekurencyjna i zostanie zgłoszony wyjątek. Jeżeli atrybutowi `age` zostanie przypisana wartość poza klasą, wtedy zarówno zapis z kropką, jak i z wbudowaną metodą `setattr` spowoduje błąd:

```
self.age = value + 10                      # Pętla
setattr(self, attr, value + 10)      # Pętla (atrbut 'age')
```

Przypisanie wartości do innego atrybutu klasy również skutkuje rekurencyjnymi wywołaniami metody `__setattr__`, jednak ten przypadek kończy się mniej dramatycznie zgłoszeniem wyjątku `AttributeError`:

```
self.other = 99                            # Rekurencja, ale bez pętli – błąd
```

Powstawianiu rekurencyjnych pętli w klasie wykorzystującej metodę `__setattr__` można zapobiec, kierując wszystkie operacje przypisania do klasy nadzędnej, zamiast przypisywać klucze w słowniku `__dict__`:

```
self.__dict__[attr] = value + 10            # OK, pętla nie powstaje
object.__setattr__(self, attr, value + 10)  # OK, pętla nie powstaje
(tylko klasy w nowym stylu)
```

Ponieważ forma z obiektem wymaga w wersji 2.x użycia klasy w nowym stylu, odłożymy szczegółowo tej operacji do rozdziału 38., w którym dokładniej przyjrzymy się zarządzaniu

atrybutami.

Argumentem trzeciej metody do zarządzania atrybutami, `__delattr__`, jest ciąg znaków zawierający nazwę atrybutu. Metoda jest wywoływana przy każdym usuwaniu atrybutu (np. za pomocą instrukcji `del obiekt.atrybut`). Podobnie jak w przypadku metody `__setattr__`, aby zapobiec powstawaniu rekurencyjnych pętli, należy użyć słownika `__dict__` lub kierować operacje usuwania atrybutów do klasy nadzędnej.



Jak się dowiemy w rozdziale 32., atrybuty zaimplementowane za pomocą funkcjonalności klas w nowym stylu, np. sloty lub właściwości, nie są fizycznie przechowywane w słowniku `__dict__` przestrzeni nazw instancji (w przypadku slotów może go nawet w ogóle nie być!). Z tego powodu w kodzie, w którym trzeba obsługiwać takie atrybuty, należy stosować pokazany wcześniej zapis `object.__setattr__`, a nie `self.__dict__`, chyba że wiadomo na pewno, że dana klasa przechowuje wszystkie swoje dane w instancji. W rozdziale 38. dowiemy się również, że podobne wymagania ma metoda `__getattribute__` w klasie w nowym stylu. Zmiana ta jest obowiązkowa w wersji Pythona 3.x, jak również w 2.x, jeżeli stosowane są klasy w nowym stylu.

Inne narzędzia do zarządzania atrybutami

Opisane trzy metody przeciążania operatorów pozwalają nam kontrolować lub specjalizować dostęp do atrybutów w obiektach. Zazwyczaj pełnią one bardzo wyspecjalizowane role; część z nich omówimy w dalszej części książki. Inny przykład użycia metody `__getattr__` jest opisany w rozdziale 28. i zawarty w pliku `person-composite.py`. Warto zapamiętać, że w Pythonie istnieje kilka sposobów zarządzania dostępem do atrybutów:

- Metoda `__getattribute__` przechwytuje wszystkie próby dostępu do atrybutów, nie tylko niezdefiniowanych. W przypadku jej użycia należy wykazać więcej ostrożności niż w przypadku `__getattr__`, aby uniknąć zapętleń.
- Funkcja wbudowana `property` pozwala powiązać z określona nazwą metody realizujące odczyt i zapis *określonego* atrybutu klasy.
- *Deskryptory* udostępniają protokół wiążący metody klasy `__get__` i `__set__` z dostępem do *określonego* atrybutu klasy.

Wymienione metody należą do zaawansowanych technik programowania i nie wszyscy programiści Pythona korzystają z nich na co dzień. Z tego powodu ich omówienie odłożymy do rozdziału 32., a szczegółową analizę technik zarządzania atrybutami do rozdziału 38.

Emulowanie prywatności w atrybutach instancji

Poniższy kod (plik `private0.py`), prezentujący użycie opisanych wcześniej narzędzi, uogólnia poprzedni przykład w celu zezwolenia każdej klasie podrzędnej na posiadanie własnej listy zmiennych prywatnych, które nie mogą być przypisane do jej instancji. (Wykorzystuje również niestandardową klasę wyjątku, na którą trzeba poczekać do części VII).

```
class PrivateExc(Exception): pass # Więcej o wyjątkach później

class Privacy:

    def __setattr__(self, attrname, value): # Dla self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self) # Zgłoszenie niestandardowego
                                             wyjątku
```

```

else:
    self.__dict__[attrname] = value      # Klucz słownika zapobiegający
zapętleniu

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']

def __init__(self):
    self.__dict__['name'] = 'Amadeusz'      # Lepsze rozwiązanie w
rozdiale 39.

if __name__ == '__main__':
    x = Test1()
    y = Test2()
    x.name = 'Edward'                      # Działa
    y.name = 'Ernest'                      # Nie działa
    print(x.name)
    y.age = 30                             # Działa
    x.age = 40                             # Nie działa
    print(y.age)

```

Tak naprawdę jest to pierwsze podejście do implementacji *prywatności zmiennych* w Pythonie (czyli niepozwolenia na modyfikację nazw atrybutów poza klasą). Choć Python nie obsługuje deklaracji prywatności jako takich, takie techniki mogą emulować dużą część tej koncepcji.

Jest to jednak rozwiązanie częściowe, a nawet toporne, dlatego aby było bardziej efektywne, musi zostać rozszerzone tak, by klasy mogły przypisywać wartości swoim atrybutom w sposób bardziej naturalny, bez odwoływanego się za każdym razem do słownika `__dict__`, co w tej chwili robi konstruktor, aby zapobiegać wywoływaniu metody `__setattr__` i zgłaszaniu wyjątku. Lepszym i pełniejszym rozwiązaniem byłoby użycie klasy opakowującej (pośredniczącej) kontrolującej dostęp do atrybutów prywatnych spoza klasy, oraz użycie metody `__getattr__` do sprawdzania operacji odczytu atrybutów.

Bardziej kompletne rozwiązanie ochrony prywatności odłożymy do rozdziału 39., w którym w sposób ogólny użyjemy *dekoratorów klas* do przechwytywania i analizy atrybutów. Choć można dzięki temu emulować prywatność, w praktyce prawie nigdy się tak nie robi. Programiści Pythona potrafią pisać duże platformy oraz aplikacje zorientowane obiektywnie bez deklaracji prywatności — co jest ciekawym odkryciem dotyczącym ogólnej kontroli dostępu, wykraczającym poza zakres naszych aktualnych celów.

Przechwytywanie referencji do atrybutów oraz przypisań jest przydatną techniką. Obsługuje *delegację* — technikę projektowania pozwalającą obiektom kodu kontrolującego na opakowywanie osadzonych obiektów, dodawanie nowego zachowania i przekierowywanie innych operacji z powrotem do opakowanych obiektów (więcej informacji na temat delegacji oraz klas opakowujących znajdziesz się w następnym rozdziale).

Reprezentacje łańcuchów – `_repr_` oraz `_str_`

Kolejne opisane tu metody służą do formatowania wyświetlanych informacji. Temu tematowi były poświęcone poprzednie rozdziały, ale tutaj go podsumujemy i sformalizujemy. Dla przypomnienia poniższy przykład wykorzystuje opisanego wcześniej konstruktora `_init_` oraz metodę przeciążania operatorów `_add_`. Użyty został również znak dodawania `+`, będący miejscowym operatorem, w celu pokazania, że też można go przeciążać (jednak zgodnie z rozdziałem 27. preferowana jest nazwana metoda). Jak już wiemy, domyślny sposób wyświetlania obiektów instancji nie jest ani praktyczny, ani ładny:

```
>>> class adder:  
...     def __init__(self, value=0):  
...         self.data = value          # Inicjalizacja zmiennej data  
...     def __add__(self, other):  
...         self.data += other        # Dodanie zmiennej other w  
miejscu  
...  
>>> x = adder()                  # Domyślne wyświetlanie  
>>> print(x)  
<__main__.adder object at 0x00000000029736D8>  
>>> x  
<__main__.adder object at 0x00000000029736D8>
```

Ale napisanie własnej metody reprezentacji łańcucha znaków lub jej odziedziczenie pozwala na dostosowanie wyświetlania do własnych potrzeb, jak w poniższym przykładzie, w którym zdefiniowana jest metoda `_repr_` zwracająca reprezentację łańcucha znaków dla instancji.

```
>>> class addrepr(adder):          # Odziedziczenie metod __init__  
oraz __add__  
...     def __repr__(self):        # Dodanie reprezentacji łańcucha  
znaków  
...         return 'addrepr(%s)' % self.data    # Konwersja na łańcuch znaków  
jako kod  
...  
>>> x = addrepr(2)                # Wykonuje metodę __init__  
>>> x + 1                        # Wykonuje metodę __add__  
(lepsza x.add?)  
>>> x                            # Wykonuje metodę __repr__  
addrepr(3)  
>>> print x                      # Wykonuje metodę __repr__  
addrepr(3)
```

```
>>> str(x), repr(x)                                # W obydwu przypadkach wykonuje
metodę __repr__
('addrepr(3)', 'addrepr(3)')
```

Po zdefiniowaniu metoda `__repr__` (i jej krewniak — metoda `__str__`) wywoływana jest automatycznie, kiedy instancje klas są wyświetlane lub przekształcane na łańcuchy znaków. Metody te pozwalają na zdefiniowanie lepszego formatu wyświetlania dla obiektów w porównaniu z domyślnym wyświetlaniem instancji. Tutaj metoda `__repr__` wykorzystuje podstawowe formatowanie ciągu w celu przekształcenia zarządzanego obiektu `self.data` na bardziej czytelny dla człowieka tekst.

Po co nam dwie metody wyświetlania?

To, co do tej pory widzieliśmy, to był ogólny obraz. Opisane metody są generalnie proste w użyciu, jednak ich role i działanie mają subtelny wpływ zarówno na projekt, jak i kod programu. W szczególności dotyczy to dwóch metod wyświetlających informacje w różny sposób, przeznaczonych do odmiennych zastosowań:

- Instrukcja `print` czy wbudowana funkcja `str` (wewnętrzna funkcja wykorzystywana przez `print`) próbują w pierwszym rzędzie wykorzystać metodę `__str__` zwracającą ciąg znaków przyjazny dla użytkownika.
- Metoda `__repr__` wykorzystywana jest we wszystkich innych przypadkach: do zwracania wartości w sesji interaktywnej, wyniku wywołania funkcji `repr`, jak również funkcji `print` i `str` w sytuacji, gdy obiekt nie implementuje metody `__str__`. Metoda `__repr__` powinna z reguły zwracać łańcuch znaków przypominający kod źródłowy, który można wykorzystać do odtworzenia obiektu oraz jako informację dla programistów, zawierającą dodatkowe informacje o obiekcie.

W skrócie: metoda `__repr__` jest wykorzystywana wszędzie, z wyjątkiem wywołań funkcji `print` i `str` w sytuacji, gdy zdefiniowana jest metoda `__str__`. Oznacza to, że można zakodować metodę `__repr__` w celu zdefiniowania formatu wyświetlania danych, którego można używać wszędzie. Można też zakodować metodę `__str__` obsługującą wyłącznie funkcje `print` i `str` lub umożliwiającą wyświetlanie informacji w alternatywny sposób.

Jak wspomniano w rozdziale 28., ogólne narzędzia też preferują metodę `__str__`, aby pozostawić innym klasom możliwość dodawania alternatywnego wyświetlania danych za pomocą metody `__repr__` w innych kontekstach, o ile funkcje `print` i `str` są wystarczające. I odwrotnie: ogólne narzędzie, które koduje metodę `__repr__`, nadal pozostawia klientom możliwość dodawania alternatywnych sposobów wyświetlania informacji za pomocą metody `__str__` w funkcjach `print` i `str`. Innymi słowy, kodując jedną z metod, pozostawia się drugą na potrzeby dodatkowego wyświetlania. W sytuacjach, gdy wybór nie jest jasny, preferowana jest metoda `__str__` do wyświetlania danych w sposób przyjazny dla użytkownika, a metoda `__repr__` do wyświetlania niskopoziomowego lub przypominającego kod źródłowy.

Napiszmy teraz kod ilustrujący bardziej konkretnie różnice między obiema metodami. W poprzednim przykładzie w tym podrozdziale metoda `__repr__` była stosowana jako metoda rezerwowa w wielu różnych kontekstach. Jednak mimo że jest ona wykorzystywana w przypadku, gdy nie ma metody `__str__`, odwrotna zasada nie obowiązuje. W innych kontekstach, np. interaktywnych poleceniach, wykorzystywana jest tylko metoda `__repr__`, a `__str__` nie jest sprawdzana w ogóle:

```
>>> class addstr(addr):
...     def __str__(self):                      # __str__, ale bez __repr__
...         return '[Wartość: %s]' % self.data    # Konwersja na ładny łańcuch
znaków
```

```

...
>>> x = addstr(3)
>>> x + 1
>>> x                                # Domyślnie metoda __repr__
<__main__.addstr object at 0x00000000029738D0>
>>> print x                           # Wykonuje metodę __str__
[Wartość: 4]
>>> str(x), repr(x)
(['[Wartość: 4]', '<__main__.addstr object at 0x00000000029738D0>'])

```

Z tego powodu metoda `__repr__` może być najlepsza, jeśli chcemy mieć *jeden* sposób wyświetlania dla wszystkich kontekstów. Definiując jednak obie metody, możemy obsługiwać odmienne sposoby wyświetlania w różnych kontekstach — na przykład wyświetlanie przeznaczone dla użytkownika obsługiwane przez `__str__` i wyświetlanie przeznaczone dla programistów w czasie ich pracy, dostępne za pomocą `__repr__`. Metoda `__str__` po prostu nadpisuje metodę `__repr__` w bardziej przyjaznych dla użytkownika kontekstach wyświetlania informacji.

```

>>> class addboth(adder):
...     def __str__(self):
...         return '[Wartość: %s]' % self.data      # Łąćuch znaków przyjazny
dla użytkownika
...     def __repr__(self):
...         return 'addboth(%s)' % self.data        # Łąćuch znaków jako kod
...
>>> x = addboth(4)
>>> x + 1
>>> x                                # Wykonuje metodę __repr__
addboth(5)
>>> print x                           # Wykonuje metodę __str__
[Wartość: 5]
>>> str(x), repr(x)
(['[Wartość: 5]', 'addboth(5)'])

```

Uwagi dotyczące wyświetlania

Opisane metody są proste w użyciu, ale w tym miejscu warto przekazać trzy użyteczne wskazówki. Po pierwsze należy pamiętać, że zarówno `__str__`, jak i `__repr__` muszą zwracać *łańcuchy znaków*; inne typy nie są obsługiwane i spowodują wyświetlanie błędów, w razie potrzeby należy je przekonwertować (np. za pomocą funkcji `str` lub znaku `%`).

Po drugie w zależności od logiki konwersji ciągów znaków kontenera metoda `__str__` jest używana wyłącznie w przypadku, gdy obiekt pojawia się na najwyższym poziomie operacji

wyświetlania. Obiekty zagnieździone w większych obiektach też mogą wyświetlać dane za pomocą własnych metod domyślnych lub metody `__repr__`. Poniższy listing demonstruje obydwa przypadki.

```
>>> class Printer:  
...     def __init__(self, val):  
...         self.val = val  
...     def __str__(self):          # Używany dla instancji  
...         return str(self.val)    # Przekształcenie na ciąg znaków  
...  
>>> objs = [Printer(2), Printer(3)]  
>>> for x in objs: print(x)           # __str__ jest wywoływane przy  
wyświetlaniu instancji,  
...                                         # ale nie w przypadku, gdy  
instancja jest elementem listy!  
2  
3  
>>> print(objs)  
[<__main__.Printer object at 0x000000000297AB38>, <__main__.Printer  
obj...itd...>]  
>>> objs  
[<__main__.Printer object at 0x000000000297AB38>, <__main__.Printer  
obj...itd...>]
```

Aby mieć pewność, że zdefiniowane wyświetlanie działa we wszystkich kontekstach niezależnie od kontenera, należy używać metody `__repr__`, nie `__str__`. Pierwsza z nich działa we wszystkich przypadkach, w których druga nie ma zastosowania.

```
>>> class Printer:  
...     def __init__(self, val):  
...         self.val = val  
...     def __repr__(self):          # __repr__ wywoływane przez  
print, jeśli nie ma __str__  
...         return str(self.val)    # __repr__ wywoływane w konsoli i  
przy zagnieżdżeniach  
...  
>>> objs = [Printer(2), Printer(3)]  
>>> for x in objs: print(x)           # Nie ma __str__: wywołuje  
__repr__  
...  
2  
3
```

```

>>> print(objs)                                # Wywołuje __repr__, nie __str__
[2, 3]
>>> objs
[2, 3]

```

Trzecia i prawdopodobnie najbardziej subtelna wskazówka dotyczy metod, które w rzadkich kontekstach mogą tworzyć nieskończone *rekurencyjne pętle*. Ponieważ niektóre obiekty zawierają metody wyświetlające inne obiekty, nie ma możliwości, aby taka metoda wywołała metodę wyświetlającą zawartą w obiekcie wyświetlonym, tworząc w ten sposób pętlę. Jest to jednak na tyle rzadki i nietypowy przypadek, że możemy go tutaj pominąć. Warto jednak zapoznać się z przykładem potencjalnego zapętlenia zawartym w uwadze umieszczonej pod koniec następnego rozdziału, opisującej metodę `__repr__` tworzącą pętlę w klasie zawartej w pliku `listinherited.py`.

W praktyce metoda `__str__` (oraz jej krewniak, metoda `__repr__`) wydaje się drugą najczęściej wykorzystywaną metodą przeciążania operatorów stosowaną w skryptach napisanych w Pythonie, zaraz za `__init__`. Za każdym razem, gdy możemy wyświetlić obiekt i zobaczyć własny sposób wyświetlania, najprawdopodobniej w użyciu jest jedna z tych metod. Więcej przykładów użycia opisanych tu narzędzi i wynikających stąd kompromisów jest zawartych w studium przypadku w rozdziale 28., w przedstawionym w rozdziale 31. kodzie wyświetlającym klasy mieszane oraz w rozdziale 35. poświęconym klasom wyjątków, gdzie wymagana jest metoda `__str__`, a nie `__repr__`.

Dodawanie prawostronne i miejscowa modyfikacja: metody `__radd__` i `__iadd__`

Kolejna grupa metod przeciążających rozszerza funkcjonalności operatorów binarnych, które już widzieliśmy, takich jak `__add__` i `__sub__` (wywoływanych za pomocą operatorów + i -). Jak wspomniałem wcześniej, jednym z powodów, dla których metod przeciążających jest tak wiele, jest mnogość odmian operatorów. Każdy operator binarny ma wariant *lewy*, *prawy* i *miejscowy*. Nawet jeżeli nie zakoduje się wszystkich trzech i stosowany jest wariant domyślny, role obiektu określają, ile wariantów należy zaimplementować.

Dodawanie prawostronne

Na przykład metoda `__add__`, która pojawiła się w poprzednim przykładzie, nie obsługuje użycia obiektów instancji po prawej stronie operatora +:

```

>>> class Adder:
    def __init__(self, value=0):
        self.data = value
    def __add__(self, other):
        return self.data + other

>>> x = Adder(5)
>>> x + 2

```

```
>>> 2 + x
TypeError: unsupported operand type(s) for +: 'int' and 'Adder'
```

By zaimplementować bardziej ogólne wyrażenia i tym samym obsługiwać operatory *przemienne*, należy zapisać w kodzie klasy również metodę `__radd__`. Python wywołuje tę metodę tylko wtedy, gdy obiekt po prawej stronie operatora `+` jest naszą instancją klasy, jednak obiekt z lewej strony nie jest instancją naszej klasy. We wszystkich innych przypadkach dla obiektu po lewej stronie wywoływana jest zamiast tego metoda `__add__`. (Wszystkie pięć opisanych w tym podrozdziale klas `Commuter` wraz z kodem samotestującym jest zawartych w pliku `commuter.py`).

```
class Commuter1:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__(self, other):
        print('radd', self.val, other)
        return other + self.val
>>> from commuter import Commuter1
>>> x = Commuter1(88)
>>> y = Commuter1(99)
>>> x + 1                                # __add__: instancja +
nieinstancja
add 88 1
89
>>> 1 + y                                # __radd__: nieinstancja +
instancja
radd 99 1
100
>>> x + y                                # __add__: instancja +
instancja, wywołuje __radd__
add 88 <__main__.Commuter1 object at 0x00000000029B39E8>
radd 99 88
187
```

Warto zwrócić uwagę na odwrócenie kolejności w metodzie `__radd__`. Atrybut `self` znajduje się tak naprawdę po prawej stronie operatora `+`, natomiast `other` po lewej. Warto również zauważyć, że `x` oraz `y` są tutaj instancjami tej samej klasy. Kiedy w wyrażeniach mieszanych pojawiają się instancje różnych klas, Python preferuje klasę instancji znajdującej się po lewej stronie. Gdy dodajemy do siebie dwie instancje, Python wywołuje metodę `__add__`, która z kolei wywołuje `__radd__`, upraszczając lewy operand.

Stosowanie metody `__add__` w `w__radd__`

Aby zaimplementować w pełni przemienną operację, która nie wymaga specjalnego uwzględniania pozycji operandów, wystarczy czasami zastosować metodę `__add__` w metodzie `radd__`. Można to osiągnąć, wywołując metodę `__add__` bezpośrednio, lub zamienić operandy miejscami i ponownie je dodać, wywołując w ten sposób metodę `__add__` pośrednio. Innym rozwiązaniem jest po prostu przypisanie metodzie `__add__` aliasu `radd__` na najwyższym poziomie instrukcji `class` (tj. w zakresie klasy). Poniżej zaimplementowane są wszystkie trzy sposoby. Za każdym razem zwracany jest ten sam wynik co na początku. W ostatnim przykładzie wykonywane jest jedno wywołanie mniejszej, dzięki czemu kod jest szybszy. We wszystkich trzech metoda `radd__` jest wywoływana wtedy, gdy atrybut `self` znajduje się po prawej stronie znaku dodawania.

```
class Commuter2:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__(self, other):
        return self.__add__(other)      # Jawne wywołanie __add__
class Commuter3:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__(self, other):
        return self + other           # Zamiana miejscami i ponowne dodanie
class Commuter4:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    __radd__ = __add__              # Alias: usunięcie pośrednika
```

Za każdym razem instancja znajdująca się po prawej stronie wywołuje jedną, wspólniezieloną metodę. Atrybutowi `self` jest przypisywany prawy operand, aby był traktowany tak samo jak lewy. Aby dokładniej poznać działanie powyższego kodu, należy samodzielnie go uruchomić. Zwracany wynik jest zawsze taki sam jak oryginalny.

Eskalowanie typu klasy

Sytuacja nieco się komplikuje w bardziej realistycznych przypadkach, gdy typ klasy powinien być przeniesiony na wynik. Do realizacji zadania należy wprowadzić sprawdzanie typów operandów, aby przekonać się, czy operację można wykonać oraz uniknąć zagnieźdżeń. Na przykład gdyby w poniższym listingu, w którym wywołana jest metodę `__add__`, która wywołuje `__radd__`, pominąć sprawdzanie typów, moglibyśmy doprowadzić do sytuacji, w której jedna instancja klasy `Commuter5` otrzymałaby wartość `val` drugiej instancji:

```
class Commuter5:                                # Przeniesienie klasy operandu na
    wynik

    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        if isinstance(other, Commuter):      # Sprawdzenie typu w celu
            uniknięcia zagnieźdżenia obiektów
                other = other.val

        return Commuter(self.val + other) # Else i wynik drugiej instancji
                                         Commuter5

    def __radd__(self, other):
        return Commuter(other + self.val)

    def __str__(self):
        return '<Commuter: %s>' % self.val

>>> x = Commuter(88)
>>> y = Commuter(99)
>>> print(x + 10)                            # Wynik jest instancją klasy
                                         Commuter5
<Commuter: 98>
>>> print(10 + y)
<Commuter: 109>
>>> z = x + y                                # Niezagnieźdżone: nie wywołuje
                                         __radd__ rekurencyjnie
>>> print(z)
<Commuter: 187>
>>> print(z + 10)
<Commuter: 197>
>>> print(z + z)
<Commuter: 375>
```

Potrzeba sprawdzania typu instancji za pomocą funkcji `isinstance` jest tutaj bardzo subtelna. Aby się o tym przekonać, należy zamienić test w komentarz, uruchomić kod i prześledzić jego przebieg. Okaże się, że w ostatniej części powyższego testu są tworzone różne zagnieźdżone obiekty, które poprawnie wykonują operacje matematyczne, ale inicjują niepotrzebne

rekurencyjne wywołania w celu uproszczenia wartości i dodatkowo wywołują konstruktory tworzące wyniki:

```
>>> z = x + y                                # Test instancji zamieniony w
komentarz

>>> print(z)

<Commuter5: <Commuter5: 187>>

>>> print(z + 10)

<Commuter5: <Commuter5: 197>>

>>> print(z + z)

<Commuter5: <Commuter5: <Commuter5: <Commuter5: 374>>>

>>> print(z + z + 1)

<Commuter5: <Commuter5: <Commuter5: <Commuter5: 375>>>
```

Pozostała część pliku *commuter.py* wygląda i działa tak jak niżej — klasy w naturalny sposób pojawiają się w krotkach:

```
#!python

from __future__ import print_function # Kompatybilność z wersjami 2.x i 3.x
...Tu zdefiniowane są klasy...

if __name__ == '__main__':
    for klass in (Commuter1, Commuter2, Commuter3, Commuter4, Commuter5):
        print('-' * 60)
        x = klass(88)
        y = klass(99)
        print(x + 1)
        print(1 + y)
        print(x + y)

c:\code> commuter.py
-----
add 88 1
89
radd 99 1
100
add 88 <__main__.Commuter1 object at 0x000000000297F2B0>
radd 99 88
187
-----
...itd...
```

Wariantów kodu jest zbyt wiele, aby je wszystkie tutaj przeanalizować, dlatego warto samodzielnie poeksperymentować z powyższymi klasami, aby je dokładniej poznać. Na przykład definiując alias `__add__` dla metody `__radd__` w klasie `Commuter5`, można oszczędzić jeden wiersz kodu, co nie uchroni jednak przed zagnieżdżeniem obiektów, jeżeli nie używa się funkcji `isinstance`. Warto zapoznać się również z zawartym w dokumentacji do Pythona opisem innych opcji. Na przykład klasy też mogą zwracać specjalny obiekt `NotImplemented`, jeżeli zostanie użyty nieobsługiwany operand, wpływając w ten sposób na wybór metody (taki przypadek jest traktowany tak, jakby metoda nie była zdefiniowana).

Dodawanie w miejscu

Aby obiekt obsłużył operację dodawania w miejscu `+=`, należy zaimplementować metodę `__iadd__` lub `__add__`. Ta druga jest stosowana w przypadku, gdy klasa nie obsługuje pierwnej. Klasa `Commuter` omawiana w poprzednim punkcie miała zaimplementowaną tę metodę właśnie z myślą o operacji `+=`, ale `__iadd__` oferuje wydajniejsze modyfikacje w miejscu:

```
>>> class Number:  
...     def __init__(self, val):  
...         self.val = val  
...     def __iadd__(self, other):          # __iadd__ wywołuje: x += y  
...         self.val += other             # Z reguły zwraca self  
...         return self  
...  
>>> x = Number(5)  
>>> x += 1  
>>> x += 1  
>>> x.val  
7
```

Jeżeli obiekty są mutowalne, powyższa metoda często szybciej dokonuje zmian w miejscu:

```
>>> y = Number([1])                      # Zmiana w miejscu jest  
szybsza niż dodawanie  
>>> y += [2]  
>>> y += [3]  
>>> y.val  
[1, 2, 3]
```

Zwykła metoda `__add__` jest uruchamiana w rezerwie i nie optymalizuje zmian w miejscu:

```
>>> class Number:  
...     def __init__(self, val):  
...         self.val = val  
...     def __add__(self, other):          # __add__: x = (x + y)
```

```

...           return Number(self.val + other)      # Przeniesienie typu na wynik
...
>>> x = Number(5)
>>> x += 1
>>> x += 1                                         # Operator += nie scala tutaj
znaków
>>> x.val
7

```

W tym podrozdziale skupiliśmy się na dodawaniu, jednak należy pamiętać, że każdy operator dwuelementowy wykorzystuje podobne metody do obsługi operacji prawostronnych oraz operacji w miejscu (na przykład `__mul__`, `__rmul__` i `__imul__`). Metody prawostronne stanowią zaawansowane zagadnienie i są dość rzadko stosowane, ponadto są konieczne wyłącznie w przypadku, gdy operatory muszą być naprzemienne. Zapewne potrzebowalibyśmy takich metod na przykład w klasie `Vector`, ale wydają się one zbędne na przykład w klasie `Employee` lub `Button`.

Wywołania — `__call__`

Metoda `__call__` wywoływana jest, kiedy wywołana zostaje nasza instancja. Nie jest to jednak definicja cykliczna — po zdefiniowaniu Python wykonuje metodę `__call__` dla wyrażeń wywołania funkcji stosowanych do naszych instancji, przekazując otrzymane argumenty pozycyjne i argumenty ze słowami kluczowymi. Dzięki temu instancja jest zgodna z opartym na funkcjach interfejsem API:

```

>>> class Callee:
...     def __call__(self, *pargs, **kargs):      # Przechwytuje wywołania
instancji
...         print('Wywołanie:', pargs, kargs)      # Akceptuje dowolne
argumenty
...
>>> C = Callee()
>>> C(1, 2, 3)                                # C jest obiektem
wywoływanym
Wywołanie: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Wywołanie: (1, 2, 3) {'y': 5, 'x': 4}

```

Bardziej formalnie, wszystkie tryby przesyłania argumentów, które analizowaliśmy w rozdziale 18., są obsługiwane przez metodę `__call__`: wszystko, co jest przesyłane do instancji, zostaje przekazane do tej metody, wraz z argumentem instancji. Poniższy listing prezentuje przykłady definicji metody `__call__`.

```

class C:
    def __call__(self, a, b, c=5, d=6): ...      # Argumenty zwykłe i domyślne

```

```

class C:
    def __call__(self, *pargs, **kargs): ... # Dowolne argumenty
class C:
    def __call__(self, *pargs, d=6, **kargs): ... # Argumenty mogące być
tylko słowami kluczowymi w 3.x

```

Powyższe definicje pozwalają wykonać następujące wywołania:

```

X = C()
X(1, 2)                                     # Pominięcie argumentów
domyślnych
X(1, 2, 3, 4)                                # Argumenty pozycyjne
X(a=1, b=2, d=4)                                # Argumenty ze słowami
kluczowymi
X(*[1, 2], **dict(c=3, d=4))                 # Rozpakowanie dowolnych
argumentów
X(1, *(2,), c=3, **dict(d=4))                # Tryb mieszany

```

Argumenty funkcji zostały opisane w rozdziale 18. W efekcie klasy i instancje zawierające metodę `__call__` obsługują tę samą składnię i semantykę argumentów, co zwykłe funkcje i metody.

Opisany sposób przechwytywania wyrażeń wywołania funkcji pozwala instancjom klas emulować wygląd i zachowanie takich obiektów jak funkcje, ale z dodatkowymi możliwościami, jak zachowanie stanu między wywołaniami (podobne możliwości mieliśmy okazję zaobserwować przy okazji omawiania zakresów w rozdziale 17., ale w niniejszym rozdziale Czytelnik powinien lepiej rozumieć działanie mechanizmu przeciążania operatorów).

```

>>> class Prod:
...     def __init__(self, value):           # Przyjmuje jeden argument
...         self.value = value
...     def __call__(self, other):
...         return self.value * other
...
>>> x = Prod(2)                         # "Zapamiętuje" wartość 2
>>> x(3)                               # 3 (przekazane) * 2
(zapamiętane)
6
>>> x(4)
8

```

W tym przykładzie `__call__` może się wydawać zbędna. Podobne narzędzie może udostępniać prosta metoda.

```

>>> class Prod:
...     def __init__(self, value):

```

```

...     self.value = value
...     def comp(self, other):
...         return self.value * other
...
>>> x = Prod(3)
>>> x.comp(3)
9
>>> x.comp(4)
12

```

Metoda `__call__` staje się jednak bardziej przydatna przy interfejsach API (np. w bibliotekach) oczekujących funkcji. Pozwala nam na tworzenie w kodzie obiektów zgodnych z oczekiwany interfejsem wywołania funkcji i przechowujących również informacje o stanie. Tak naprawdę jest to chyba trzecia najczęściej wykorzystywana metoda przeciążania operatorów, po konstruktorze `__init__` oraz alternatywnych metodach formatu wyświetlanego `__str__` oraz `__repr__`.

Interfejsy funkcji i kod oparty na wywołaniach zwrotnych

Jako przykład weźmy wspomniany wcześniej zbiór narzędzi tkinter graficznego interfejsu użytkownika (noszący w Pythonie 2.x nazwę Tkinter), który pozwala na rejestrowanie funkcji jako programów obsługi zdarzeń (ang. *event handler*, inaczej wywołań zwrotnych, ang. *callback*). Kiedy następuje zdarzenie, tkinter wywołuje zarejestrowane obiekty. Jeśli chcemy, by program obsługi zdarzeń zachowywał stan pomiędzy zdarzeniami, możemy albo zarejestrować wiązaną metodę klasy, albo instancję zgodną z oczekiwany interfejsem z `__call__`.

W kodzie z poprzedniego podrozdziału zarówno `x.comp` z drugiego przykładu, jak i `x` z pierwszego mogą w ten sposób zostać uznane za obiekty podobne do funkcji. Podobny efekt można uzyskać za pomocą opisanych w rozdziale 17. domknięć. Nie obsługują one jednak tak dobrze zwielokrotnionych operacji ani nie można ich dostosowywać.

Więcej informacji na temat metod z wiązaniem znajdzie się w kolejnym rozdziale, natomiast poniżej znajduje się hipotetyczny przykład zastosowania metody `__call__` w dziedzinie graficznych interfejsów użytkownika. Poniższa klasa definiuje obiekt obsługujący interfejs wywołania funkcji, ale również posiada informacje o stanie zapamiętujące kolor, na jaki po naciśnięciu powinien zmieniać się przycisk.

```

class Callback:
    def __init__(self, color):                      # Funkcja i informacje o stanie
        self.color = color
    def __call__(self):                            # Obsługa wywołań bez argumentów
        print('włącz', self.color)

```

W kontekście graficznego interfejsu użytkownika możemy zarejestrować instancje tej klasy jako programy obsługi zdarzeń przycisków, nawet jeśli GUI oczekuje, że programy obsługi zdarzeń będzie można wywołać jak proste funkcje bez argumentów.

```

# Kod obsługujący zdarzenia
cb1 = Callback('niebieski')                      # 'Pamięta' niebieski
cb2 = Callback('zielony')                         # 'Pamięta' zielony
B1 = Button(command=cb1)                           # Rejestrowanie programów
obsługi
B2 = Button(command=cb2)                           # Rejestrowanie programów
obsługi

```

Po późniejszym naciśnięciu przycisku instancja obiektu wywoływana jest jako prosta funkcja bez argumentów — dokładnie tak, jak w poniższych wywołaniach. Ponieważ zachowuje ona stan jako atrybuty instancji, pamięta również, co ma zrobić. Staje się tzw. funkcją stanową:

```

# Zdarzenia
cb1()                                              # Wyświetla 'włącz niebieski'
cb2()                                              # Wyświetla 'włącz zielony'

```

Tak naprawdę jest to chyba najlepszy sposób zachowania informacji o stanie w Pythonie (przynajmniej zgodnie z obowiązującymi w tym języku zasadami). Dzięki programowaniu zorientowanemu obiektywnie zapamiętany stan jest jawnym dzięki użyciu przypisania atrybutów. Jest to technika odmienna od opisanych wcześniej (zmienne globalne, referencje do zakresu funkcji zawierającej oraz domyślne argumenty zmienne), opierających się na bardziej ograniczonym lub niejawnym działaniu. Co więcej, dodawanie struktury i dostosowywanie klas obejmuje nie tylko zachowywanie stanu.

Z drugiej strony narzędzia takie jak funkcje domknięć dobrze nadają się do podstawowego przechowywania stanu, a instrukcja `nonlocal` w wersji 3.x sprawia, że otaczające zakresy stają się realną alternatywą w wielu programach. Związanego z tym kompromisu przeanalizujemy w rozdziale 39., gdy zaczniemy kodować dekoratory. Tutaj przedstawiony jest prosty odpowiednik *domknięcia*:

```

def callback(color):      # Otwierający zakres kontra atrybuty
    def oncalle():
        print('włącz', color)
    return oncalle
cb3 = callback('złoty')   # Funkcja do zarejestrowania
cb3()                     # Gdy wystąpi zdarzenie, wyświetlany jest napis
                          'włącz złoty'

```

Zanim przejdziemy dalej, powiem jeszcze, że istnieją dwa kolejne sposoby przypinania w ten sposób informacji do funkcji zwrotnych. Jedna możliwość to wykorzystanie argumentów domyślnych w funkcjach `lambda`.

```

cb4 = (lambda color='czarny': 'włącz ' + color) # Lub: wartości domyślne
print(cb4())

```

Drugą opcją jest wykorzystanie metod klasy z *wiązaniem* (będzie opisana później, ale jest na tyle prosta, że można ją tutaj zapowiedzieć). Jest to rodzaj obiektu, który pamięta instancję `self` oraz funkcję z referencji w taki sposób, że może później być wywołany jak prosta funkcja bez instancji.

```
class Callback:
```

```

def __init__(self, color):                      # Klasa z informacjami o stanie
    self.color = color
def changeColor(self):                          # Normalna nazwana metoda
    print('włącz', self.color)
cb1 = Callback('niebieski')
cb2 = Callback('zielony')
B1 = Button(command=cb1.changeColor)           # Metoda z wiązaniem:
referencja, ale bez wywołania
B2 = Button(command=cb2.changeColor)           # Pamięta funkcję i self

```

Kiedy przycisk ten zostanie później naciśnięty, zachowuje się, jakby odpowiedzialny za to był graficzny interfejs użytkownika, wywołując metodę `changeColor` w celu przetworzenia informacji o stanie obiektu, a nie o samej instancji:

```

object = Callback('niebieski')
cb = object.changeColor                      # Zarejestrowany program obsługi
zdarzeń
cb()                                         # Po wystąpieniu zdarzenia
wyświetla 'niebieski'

```

W powyższym przykładzie nie musimy używać wyrażenia `lambda`, ponieważ sama referencja do metody pozwala opóźnić jej wywołanie. Ta technika jest prostsza, jednak mniej uniwersalna od przeciążania wywołań za pomocą metody `__call__`. Przypominam, że informacje na temat metod z wiązaniem znajdują się w następnym rozdziale.

Kolejny przykład zastosowania metody `__call__` zobaczymy w rozdziale 32., w którym wykorzystamy ją do zaimplementowania czegoś, co znane jest pod nazwą *dekoratora funkcji* (ang. *function decorator*) — obiektu wywoływalnego, który dodaje warstwę logiki na wierzchu funkcji osadzonej. Ponieważ metoda `__call__` pozwala na dołączanie informacji o stanie do obiektu wywoływalnego, jest to naturalna technika implementacyjna dla funkcji, która musi pamiętać i wywołać inną funkcję. Więcej przykładów użycia metody `__call__` jest zawartych w części dotyczącej zachowywaniu stanu w rozdziale 17. oraz w rozdziałach 39. i 40. opisujących bardziej zaawansowane zagadnienia dotyczące dekoratorów i metaklas.

Porównania — `__lt__`, `__gt__` i inne

Następna grupa metod przeciążających obsługuje operacje porównywania. Jak sugeruje tabela 30.1, klasy mogą definiować metody przechwytyujące sześć operacji porównania: `<`, `>`, `<=`, `>=`, `==` i `!=`. Metody przeciążające te operatory są dość proste w użyciu, ale należy pamiętać o pewnych podstawowych zasadach:

- W przeciwnieństwie do par metod typu `__add__`/`__radd__` w przypadku porównań nie ma takich odpowiedników. Natomiast są dostępne odwrócenia operacji, jeżeli tylko jeden operand obsługuje porównanie (np. odwróceniem operacji `__lt__` jest `__gt__` i *vice versa*).
- Nie istnieją jawne powiązania pomiędzy operatorami porównania. Prawda zwrocona przez operator `==` nie oznacza, że operator `!=` na tych samych operandach zwróci fałsz, dlatego metody `__eq__` i `__ne__` powinny być zdefiniowane w sposób niezależny, aby zapewnić prawidłowe działanie operatorów.

- W Pythonie 2.x metoda `__cmp__` jest wykorzystywana przez wszystkie operacje porównania, jeśli nie są zdefiniowane odpowiednie metody dedykowane. Metoda ta zwraca liczbę mniejszą od zera, równą zeru lub większą w zależności od tego, czy pierwszy operand (`self`) jest odpowiednio mniejszy, równy drugiemu lub od niego większy. Ta metoda często używa funkcji wbudowanej `cmp(x, y)`. Metoda `__cmp__` oraz funkcja wbudowana `cmp` zostały usunięte w Pythonie 3.x, zatem jesteśmy zmuszeni do wykorzystania dedykowanych metod.

Nie mamy miejsca na dokładną analizę metod obsługujących operatory porównania, ale w ramach szybkiego wprowadzenia weźmy pod uwagę następujący kod klasy:

```
class C:
    data = 'spam'

    def __gt__(self, other):           # 3.x i 2.x
        return self.data > other

    def __lt__(self, other):
        return self.data < other

X = C()

print(X > 'ham')                  # True (wywołuje __gt__)
print(X < 'ham')                  # False (wywołuje __lt__)
```

W przypadku uruchomienia w Pythonie 3.x lub 2.x skrypt ten wypisze dla każdego porównania wyniki podane w komentarzach, ponieważ metody klasy przechwytyują i implementują operacje porównania. Więcej szczegółowych informacji na ten temat jest zawartych w podręczniku do Pythona i innych materiałach. Na przykład metoda `__lt__` jest wykorzystywana w operacji sortowania w Pythonie 3.x. Metody te, podobnie jak operatory binarne, zgłaszały wyjątek `NotImplemented` w przypadku użycia nieobsługiwanych argumentów.

Metoda `__cmp__` w 2.x

W Pythonie 2.x (tylko w tej wersji) dostępna jest metoda `__cmp__` wykorzystywana jako mechanizm zastępczy w przypadku, gdy nie jest zdefiniowana metoda dedykowana do obsługi danego operatora. Metoda ta zwraca wyniki całkowite sygnalizujące wynik porównania operandów. Poniższy kod zadziała w 2.x, ale nie zadziała w 3.x, ponieważ w tej wersji metoda `__cmp__` nie jest już wywoływana.

```
class C:
    data = 'spam'                      # Tylko 2.x

    def __cmp__(self, other):           # __cmp__ nieużywane w 3.x
        return cmp(self.data, other)    # cmp niezdefiniowane w 3.x

X = C()

print(X > 'ham')                  # True (wywołuje __cmp__)
print(X < 'ham')                  # False (wywołuje __cmp__)
```

Należy zwrócić uwagę, że powyższy kod uruchomiony w wersji Pythona 3.x zakończy się błędem, ponieważ w tej wersji nie ma specjalnej metody `__cmp__`, a nie dlatego, że nie ma wbudowanej funkcji `cmp`. Jeżeli powyższą klasę zmieni się w pokazany niżej sposób w celu

zasymulowania wywołania `cmp`, to kod będzie dalej działał w wersji 2.x, w przeciwieństwie do wersji 3.x.

```
class C:  
    data = 'spam'  
  
    def __cmp__(self, other):  
        return (self.data > other) - (self.data < other)
```

Ktoś mógłby zapytać, po co prezentuję metodę, której nie ma już w Pythonie 3.x. Z pewnością łatwiej byłoby kompletnie wymazać historię, ale niniejsza książka ma za zadanie omawiać wersje 2.x i 3.x. Operator `__cmp__` jest obsługiwany w 2.x, przez co osoby programujące w tej wersji mogą napotkać jego użycie w kodzie, który muszą obsługiwać, zatem jestem zobowiązany wspomnieć o tej zasłości historycznej. Co więcej, metoda `__cmp__` została porzucona dość nagle — w porównaniu z inną porzuconą metodą `__getslice__` omawianą wcześniej, ale była stosunkowo częściej stosowana w kodzie, przez co łatwiej się na nią natknąć. Jeśli ktoś chce zachować zgodność pisaneego kodu z Pythonem 3.x, powinien zrezygnować ze stosowania metody `__cmp__` i przepisać swój kod tak, aby używać dedykowanych metod obsługujących operacje porównań.

Testy logiczne — `__bool__` i `__len__`

Kolejny zestaw metod jest bardzo przydatny. Jak wiemy, każdy obiekt w Pythonie z definicji reprezentuje wartość *prawda* lub *fałsz*. Gdy tworzy się klasę, można zdefiniować, co to oznacza dla obiektów, kodując metody zwracające wyniki `True` lub `False`. Nazwy tych metod różnią się w zależności od wersji Pythona. W tym podrozdziale zajmiemy się najpierw metodami w wersji 3.x, a potem ich odpowiednikami w 2.x.

Jak wspomniałem wcześniej, w kontekście logicznym Python najpierw próbuje bezpośrednio uzyskać wartość logiczną, wywołując metodę `__bool__`. Jeżeli tej metody nie ma, wywołuje metodę `__len__` i określa wynik logiczny na podstawie długości obiektu. W pierwszym przypadku do utworzenia wyniku wykorzystywany jest stan obiektu lub inna informacja. W wersji 3.x kod wygląda następująco:

```
>>> class Truth:  
...     def __bool__(self): return True  
...  
>>> X = Truth()  
>>> if X: print('tak!')  
...  
tak!  
>>> class Truth:  
...     def __bool__(self): return False  
...  
>>> X = Truth()  
>>> bool(X)
```

```
False
```

Jeśli ta metoda nie jest zdefiniowana, Python próbuje określić długość obiektu: obiekty niepuste są w operacjach logicznych traktowane jako odpowiedniki wartości True (długość większa od zera implikuje wartość True, zero oznacza False).

```
>>> class Truth:  
...     def __len__(self): return 0  
...  
>>> X = Truth()  
>>> if not X: print('nie!')  
...  
nie!
```

Jeśli w klasie są zdefiniowane *obie* te metody, w kontekście logicznym Python wybierze metodę `__bool__`, ponieważ jest specjalizowana do tego typu operacji.

```
>>> class Truth:  
...     def __bool__(self): return True          # 3.x najpierw wywołuje  
__bool__  
...     def __len__(self): return 0              # 2.x najpierw wywołuje  
__len__  
...  
>>> X = Truth()  
>>> if X: print('tak!')  
...  
tak!
```

Jeśli w obiekcie nie jest zdefiniowana żadna z tych metod, obiekt zostanie zinterpretowany jako wartość True (co może mieć poważne skutki uboczne dla osób o zainteresowaniach metafizycznych).

```
>>> class Truth:  
...     pass  
...  
>>> X = Truth()  
>>> bool(X)  
True
```

Tak wygląda klasa `Truth` w wersji 3.x. Powyższe przykłady uruchomione w wersji 2.x nie zgłaszą wyjątków, ale niektóre wyniki mogą wyglądać dziwnie, jeżeli nie przeczyta się kolejnego podrozdziału.

Metody logiczne w Pythonie 2.x

Użytkownicy Pythona 2.x mogą we wszystkich przykładach w poprzednim rozdziale stosować metodę `__nonzero__` zamiast `__bool__`. W Pythonie 3.x nastąpiła zmiana nazwy metody `__nonzero__` na `__bool__`, ale operacje logiczne działają w obu wersjach tak samo. Zarówno w wersji 3.x, jak i 2.x jako rezerwa jest stosowana metoda `__len__`.

Uruchamiając w 2.x pierwszy test z poprzedniego podrozdziału, otrzymamy takie same wyniki, ale to tylko przypadek, ponieważ metoda `__bool__` nie jest rozpoznawana, a pod nieobecność metod specjalnych wszystkie obiekty zwracają wartość `True!` Aby zaobserwować różnice między wersjami, zmuśmy kod do zwracania wartości `False`:

```
C:\code> c:\python33\python

>>> class C:
...     def __bool__(self):
...         print('bool')
...         return False
...
...
>>> X = C()
>>> bool(X)
bool
False
>>> if X: print(99)
...
bool
```

W wersji 3.x kod zadziała zgodnie z oczekiwaniemi. Jednak w 2.x metoda `__bool__` jest ignorowana, co powoduje, że obiekty są interpretowane jako `True`.

```
C:\code> c:\python27\python

>>> class C:
...     def __bool__(self):
...         print('bool')
...         return False
...
...
>>> X = C()
>>> bool(X)
True
>>> if X: print(99)
...
99
```

W 2.x powinniśmy zastosować metodę `__nonzero__` albo zwrócić zero z metody `__len__`, sygnalizując wartość `False`.

```
C:\code> c:\python27\python
```

```

>>> class C:
...     def __nonzero__(self):
...         print('nie zero')
...         return False      # Zwraca liczbę całkowitą (True/False, czyli
1/0)
...
...
>>> X = C()
>>> bool(X)
nie zero
False
>>> if X: print(99)
...
nie zero

```

Należy pamiętać, że metoda `__nonzero__` działa tylko w 2.x; użyta w 3.x zostanie po cichu zignorowana, a obiekt zostanie domyślnie sklasyfikowany jako True. Podobnie rzecz się ma z metodą `__bool__` użytą w wersji 2.x.

Skoro już udało się nam zahaczyć o obszary metafizyczne, nadszedł czas, aby przejść do ostatniego zagadnienia związanego z przeciążaniem operatorów — *śmierci obiektów*.

Destrukcja obiektu — `__del__`

Nadszedł czas, aby zakończyć ten rozdział i dowiedzieć się, jak można kończyć pracę z obiektami. Jak widzieliśmy, *konstruktor `__init__`* jest wywoływany przy każdym tworzeniu instancji (a wcześniej metoda `__new__` tworząca obiekt). W Pythonie istnieje też druga strona: metoda *destruktora `__del__`* wywoływana automatycznie w momencie, gdy interpreter zwalnia pamięć zajmowaną przezinstancję (na przykład w procesie „odśmiecania”, ang. *garbage collection*).

```

>>> class Life:
...     def __init__(self, name='nieznajomy'):
...         print('Witaj', name)
...         self.name = name
...     def __del__(self):
...         print('Żegnaj', self.name)
...
...
>>> brian = Life('Brian')
Witaj Brian
>>> brian = 'loretta'
Żegnaj Brian

```

Gdy zmiennej `brian` przypisujemy ciąg znaków, tracimy ostatnią referencję do utworzonej wcześniej instancji klasy `Life`, co powoduje wywołanie destruktora. Tego typu zachowanie może być pomocne przy tworzeniu procedur porządkujących (na przykład w celu zwolnienia połączeń do serwera). Jednak z kilku powodów, opisanych w następnym podrozdziale, destruktory nie są tak powszechnie stosowane w Pythonie jak w niektórych innych językach zorientowanych obiektywnie.

Uwagi dotyczące stosowania destruktorów

Metoda destruktora działa zgodnie z dokumentacją, ale ma pewne dobrze znane osobliwości i ciemne cechy, które sprawiają, że stosuje się ją dość rzadko:

- *Potrzeba stosowania.* Po pierwsze destruktory nie są tak przydatne w Pythonie jak w innych językach obiektowych. Ponieważ Python automatycznie zwalnia pamięć zajmowaną przez nieużywane obiekty, destruktory nie są potrzebne do zarządzania pamięcią. Zgodnie z aktualną dokumentacją nie trzeba zamykać obiektów plików wykorzystywanych przez instancję, ponieważ są one zamknięte automatycznie podczas usuwania obiektu. Jednak jak wspomniałem w rozdziale 9., czasami warto wywoływać metody zamykające pliki, ponieważ automatyczne zamknięcie może przebiegać różnie w zależności od implementacji Pythona (np. w Jython).
- *Przewidywalność.* Nie zawsze jest łatwo przewidzieć, kiedy instancja może zostać zwolniona. W niektórych przypadkach do obiektu mogą istnieć dodatkowe referencje, na przykład w strukturach systemowych, które uniemożliwią wywołanie jego destruktora. Ponadto Python nie gwarantuje, że metody destruktatorów są wywoływane w obiektach, które istnieją po zamknięciu interpretera.
- *Wyjątki.* W rzeczywistości użycie metody `__del__` może sprawiać problemy z jeszcze kilku, bardziej subtelnych powodów. Wyjątki wywoływane w tej metodzie po prostu wysyłają komunikaty do standardowego strumienia błędów `sys.stderr`. Zdarzenia wyjątków nie są zgłaszane, ponieważ trudno jest określić kontekst wywołania destruktora przez mechanizm odśmiecania pamięci i nie wiadomo, gdzie takie wyjątki powinny być dostarczane.
- *Zapętlenia.* Ponadto zapętlenia referencji (referencje cykliczne), mogą całkowicie uniemożliwić odśmiecanie pamięci. Opcjonalny detektor tego typu zapętleń, który domyślnie jest aktywny i potrafi wykryć i usunąć z pamięci takie referencje, jest stosowany wyłącznie w przypadku, gdy obiekt posiada metodę `__del__`. Szczegóły są dość zagmatwane, nie będę ich zatem tu omawiał. Zainteresowanych odsyłam do standardowego podręcznika Pythona: warto zajrzeć do dokumentacji metody `__del__` oraz modułu odśmiecującego `gc`.

Z powodu powyższych mankamentów często lepiej jest kodować przerywanie aktywności za pomocą jawnie wywoływanej metody (np. `shutdown`). Można to też osiągnąć przy użyciu opisanej w następnym rozdziale instrukcji `try/finally` oraz instrukcji `with` w przypadku obiektów obsługujących model menedżera kontekstu.

Podsumowanie rozdziału

To już koniec przykładów przeciążania metod, na więcej nie mamy miejsca. Wiele innych, nieomówionych szczegółowo metod przeciążania operatorów działa w podobny sposób do tych, którym poświęciliśmy więcej uwagi. Wszystkie są natomiast jedynie punktami zaczepienia umożliwiającymi przechwytywanie wbudowanych operacji. Niektóre metody przeciążania operatorów wymagają zastosowania specjalnych list argumentów lub zwracają nietypowe wartości, jednak z reguły ze wszystkich korzysta się w taki sam sposób. Kilka przykładów będziemy mieli okazję poznać w dalszych częściach książki:

- Rozdział 34. wykorzystuje metody `__enter__` i `__exit__` w instrukcji `with` menedżera kontekstu.
- Rozdział 38. wykorzystuje metody `__get__` i `__set__` używane w deskryptorach klas do przechwycenia metod odczytujących i przypisujących wartości atrybutom.
- Rozdział 40. wykorzystuje metodę `__new__` służącą do tworzenia obiektów w kontekście metaklasy.

Dodatkowo niektóre metody omawiane w niniejszym rozdziale, jak `__call__` i `__str__`, będą dość powszechnie wykorzystywane w wielu przykładach tej książki. Szczegółowe omówienie wszystkich dostępnych metod można znaleźć w innych źródłach. Odsyłam Czytelnika do standardowego podręcznika Pythona oraz innych książek omawiających szczegółowo pozostałe metody przeciążania operatorów.

W następnym rozdziale odkładamy omawianie wewnętrznych mechanizmów klas, aby przejść do najczęściej stosowanych wzorców projektowych służących do optymalizacji ponownego użycia kodu. Potem zajmiemy się kilkoma zaawansowanymi tematami oraz ostatnim ważnym zagadnieniem w tej książce — wyjątkami. Zanim jednak przejdziemy dalej, sugeruję zatrzymać się na chwilę nad quizem do zagadnień z niniejszego rozdziału.

Sprawdź swoją wiedzę – quiz

1. Jakie dwie metody specjalne można wykorzystać do zaimplementowania iteracji w tworzonych klasach?
2. Jakie dwie metody specjalne obsługują wyświetlanie obiektów i w jakich kontekstach są stosowane?
3. W jaki sposób samodzielnie obsłużyć w klasie operację wycinania fragmentu sekwencji?
4. W jaki sposób przechwycić w klasie operację dodawania w miejscu?
5. W jakich przypadkach należy wykorzystywać mechanizm przeciążania operatorów?

Sprawdź swoją wiedzę – odpowiedzi

1. Klasy mogą implementować iterację przez zdefiniowanie (lub odziedziczenie) metody `__getitem__` lub `__iter__`. We wszystkich kontekstach iteracyjnych Python próbuje użyć metody `__iter__` (która powinna zwrócić obiekt obsługujący protokół iteracyjny udostępniający metodę `__next__`). Jeśli metoda `__iter__` nie zostanie znaleziona, Python przełącza się w tryb użycia metody indeksującej `__getitem__` (która jest wywoływana sekwencyjnie z kolejnymi indeksami elementów). W takim wypadku instrukcja `yield` automatycznie tworzy metodę `__next__`.
2. Wyświetlanie obiektów jest obsługiwane przez metody `__str__` i `__repr__`. Pierwsza z nich jest wywoływana w przypadku użycia funkcji wbudowanych `print` lub `str`, druga jest wywoływana dla funkcji `print` i `str` w przypadku, gdy obiekt nie posiada metody `__str__`, a dodatkowo zawsze w przypadku użycia funkcji wbudowanej `repr`, w konsoli interaktywnej do zwracania wartości obiektu na konsolę (`echo`) oraz w sytuacjach zagnieźdzenia obiektów. Innymi słowy, `__repr__` jest używana zawsze, z wyjątkiem użycia `print` i `str`, gdy obiekt posiada metodę `__str__`. Zwyczajowo

funkcja `__str__` powinna zwracać informacje o obiekcie w formacie czytelnym dla użytkownika, natomiast `__repr__` wyświetla szczegóły obiektu w formacie przypominającym kod źródłowy.

3. Tworzenie wycinków odbywa się przez metodę indeksującą `__getitem__`, wywołaną z obiektem wycinka zamiast zwykłego indeksu. Obiekty wycinków mogą być przetwarzane w dowolny sposób. W Pythonie 2.x dodatkowo dostępna jest metoda `__getslice__` (wycofana w 3.x) obsługująca wycinki z zadanyimi obiema granicami.
4. Operacja dodawania w miejscu próbuje wywołać metodę `__iadd__`, a jeśli jej nie znajdzie, szuka `__add__`. Taka sama zasada jest stosowana dla wszystkich operacji dwuoperatorowych. Dodatkowo dostępna jest metoda `__radd__`, służąca do dodawania argumentów po prawej stronie.
5. Przeciążanie operatorów należy stosować w sytuacjach, gdy klasa w naturalny sposób odzwierciedla lub musi emulować interfejsy wbudowanych typów danych. Na przykład kolekcje mogą emulować interfejsy sekwencji lub mapowań, a obiekty wywoływalne można kodować tak, aby można je było stosować z interfejsami API wymagającymi podania funkcji. Nie powinno się implementować operatorów wyrażeń, jeżeli w naturalny i logiczny sposób nie są one powiązane z obiektami. W takich przypadkach zaleca się stosowanie zwykłych nazwanych metod.

Rozdział 31. Projektowanie z użyciem klas

Dotychczas w tej części książki koncentrowaliśmy się na wykorzystywaniu narzędzia programowania zorientowanego obiektowego Pythona, czyli klasy. Programowanie zorientowane obiektowo to jednak również *kwestie związane z projektowaniem*, czyli to, jak wykorzystać klasy do modelowania przydatnych obiektów. W niniejszym rozdziale zajmiemy się kilkoma zagadnieniami kluczowymi dla programowania zorientowanego obiektowo i zaprezentujemy dodatkowe przykłady, bardziej realistyczne od pokazanych wcześniej.

Omówimy wiele powszechnie stosowanych w Pythonie wzorców projektowych, jak dziedziczenie, delegacja, kompozycja czy fabryki. Przeanalizujemy też koncepcje klas z punktu widzenia projektowania, omawiając atrybuty pseudoprivatne, wielokrotne dziedziczenie i metody związane.

Wiele z zawartych tu koncepcji wymaga nieco szerszego objaśnienia niż to, które jestem w stanie zamieścić w niniejszej książce. Jeśli kogoś one zaciekawią, sugeruję znalezienie jakiegoś tekstu poświęconego projektowaniu w programowaniu zorientowanym obiektowo lub wzorom projektowym. Dobra wiadomość jest taka, że — jak się przekonamy — w Pythonie wiele tradycyjnych wzorców projektowych jest trywialnie prostych.

Python a programowanie zorientowane obiektowo

Implementację programowania zorientowanego obiektowo w Pythonie można sprowadzić do trzech kwestii.

Dziedziczenie

Dziedziczenie jest w Pythonie oparte na wyszukiwaniu atrybutów (w wyrażeniach `X.nazwa`).

Polimorfizm

W wyrażeniu `X.metoda` znaczenie atrybutu `metoda` uzależnione jest od typu (klasy) obiektu `X`.

Hermetyzacja (enkapsulacja)

Metody oraz operatory implementują zachowanie. Ukrywanie danych jest domyślną konwencją.

Na tym etapie lektury każdy powinien wiedzieć już, czym jest w Pythonie dziedziczenie. Kilka razy wspominaliśmy również o polimorfizmie Pythona — wynika on z braku deklaracji typów w tym języku. Ponieważ atrybuty są zawsze interpretowane w czasie wykonywania, obiekty implementujące te same interfejsy są wymienne. Klient nie musi wiedzieć, jakie rodzaje obiektów implementują wywoływane metody.

Hermetyzacja oznacza w Pythonie pakowanie, czyli ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Nie chodzi tu o wymuszenie prywatności danych, choć i taką własność daje się osiągnąć w kodzie, o czym przekonamy się w rozdziale 39. Hermetyzacja pozwala na modyfikację implementacji interfejsu obiektu bez ingerowania w użytkowników tego obiektu.

Polimorfizm to interfejsy, a nie sygnatury wywołań

Niektóre języki zorientowane obiektowo definiują polimorfizm jako przeciążanie funkcji w oparciu o sygnatury typów argumentów. Ponieważ jednak w Pythonie nie istnieją deklaracje typów, koncepcja ta nie ma w rzeczywistości zastosowania. Polimorfizm w Pythonie oparty jest na *interfejsach* obiektów, nie na typach.

Jeżeli ktoś tęskni do czasów C++, może spróbować przeciążać metody za pomocą ich list argumentów, jak w poniższym kodzie.

```
class C:  
    def meth(self, x):  
        ...  
    def meth(self, x, y, z):  
        ...
```

Kod ten będzie działał, jednak ponieważ instrukcja `def` po prostu przypisuje obiekt do nazwy w zakresie klasy, ostatnia definicja funkcji metody będzie tą, która zostanie zachowana (to dokładnie tak samo jak wywołanie najpierw `X = 1`, a później `X = 2` — zmienna `X` będzie miała wartość 2). Zatem może być tylko jedna definicja metody.

Jeżeli naprawdę są potrzebne przecinane metody, zawsze można zakodować wybór oparty na typie, wykorzystując techniki sprawdzania typów omówione w rozdziałach 4. oraz 9. lub narzędzia list argumentów z rozdziału 18.

```
class C:  
    def meth(self, *args):  
        if len(args) == 1:                      # Sprawdzenie liczby argumentów  
            ...  
        elif type(args[0]) == int:              # Sprawdzenie typu argumentu  
            (można użyć isinstance())  
            ...
```

Normalnie nie powinniśmy tego jednak robić, ponieważ nie jest to sposób praktykowany w Pythonie. Zgodnie z informacjami z rozdziału 16. kod powinniśmy pisać w taki sposób, by oczekiwał on określonego interfejsu obiektu, a nie specyficznego *typu* danych. W ten sposób będzie przydatny dla szerszej kategorii typów oraz zastosowań, zarówno teraz, jak i w przyszłości.

```
class C:  
    def meth(self, x):  
        x.operation()                         # Zakładamy, że x robi coś  
        właściwego
```

Powszechnie uważa się również, że lepiej jest używać osobnych *nazw* metod dla różnych operacji, zamiast polegać na sygnaturach wywołania (bez względu na to, w jakim języku

programowania tworzymy kod).

Koncepcja obiektowa w Pythonie jest łatwa do zrozumienia, jednak cała sztuka programowania obiektowego polega na sposobie łączenia klas ze sobą w celu realizacji zadań pisanego programu. Kolejny podrozdział rozpocznie analizę dostępnych w Pythonie koncepcji programowania obiektowego.

Programowanie zorientowane obiektywne i dziedziczenie – związek „jest”

Omówiliśmy już dość szczegółowo mechanizmy dziedziczenia, jednak chciałbym zaprezentować przykład tego, w jaki sposób można wykorzystać dziedziczenie do modelowania relacji ze świata rzeczywistego. Z punktu widzenia programisty dziedziczenie uruchamiane jest przez kwalifikację atrybutów powodującą wyszukiwanie zmiennych w instancjach, ich klasach oraz wszystkich klasach nadzędnych. Z punktu widzenia projektanta dziedziczenie jest sposobem określania przynależności do zbioru. Klasa definiuje zbiór właściwości, które mogą być dziedziczone oraz dostosowywane przez bardziej specyficzne podzbiory (czyli klasy podrzędne).

By to zilustrować, powróćmy do naszego robota do robienia pizzy z początku tej części książki. Założymy, że zdecydowaliśmy się wybrać inną ścieżkę kariery i otwieramy pizzerię. Jednym z pierwszych naszych działań jest zatrudnienie pracowników obsługujących klientów czy przygotowujących jedzenie. Jednak ponieważ w głębi serca nadal jesteśmy inżynierami, decydujemy się zbudować robota wytwarzającego pizzę. Ze względu na poprawność polityczną oraz cybernetyczną uznajemy, że nasz robot będzie również pełnoprawnym pracownikiem z wynagrodzeniem.

Nasz zespół pracowników został w przykładowym pliku *employees.py* (działającym w wersjach Pythona 2.x i 3.x) zdefiniowany przez cztery klasy. Klasa najbardziej ogólna (*Employee*) udostępnia wspólne zachowanie, takie jak wzrost wynagrodzenia (*giveRaise*) czy reprezentację tekstową (*__repr__*). Istnieją dwa typy pracowników i tym samym dwie klasy podrzędne — *Chef*, czyli kucharz, oraz *Server*, czyli kelner. Obie przesyłają odziedziczoną metodę *work* w celu wyświetlania bardziej specyficznych komunikatów. Nasz robot wytwarzający pizzę należy do jeszcze bardziej wyspecjalizowanej klasy — *PizzaRobot* jest klasą potomną klasy *Chef*, która jest z kolei klasą potomną klasy *Employee*. W terminologii programowania zorientowanego obiektowo nazywamy te relacje *związkami typu „jest”* (ang. „is-a”) — robot jest kucharzem, który z kolei jest pracownikiem. Poniżej widać zawartość pliku *employees.py*.

```
# Plik employees.py (wersja 2.x i 3.x)

from __future__ import print_function
class Employee:

    def __init__(self, name, salary=0):
        self.name = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print(self.name, "robi różne rzeczy")
    def __repr__(self):
```

```

        return "<Pracownik: imię=%s, wynagrodzenie=%s>" % (self.name,
self.salary)

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, "przygotowuje jedzenie")

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, "obsługuje klienta")

class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, "przygotowuje pizzę")

if __name__ == "__main__":
    bob = PizzaRobot('robert')                      # Tworzy robota o imieniu
Robert
    print(bob)                                       # Wykonuje dziedziczoną
metodę __repr__
    bob.work()                                       # Wykonuje działanie
specyficzne dla typu
    bob.giveRaise(0.20)                             # Daje robotowi 20-procentową
podwyżkę
    print(bob); print()
    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()

```

Kiedy wykonujemy ten moduł jako samodzielny program, zostanie uruchomiony kod testujący, który tworzy wytwarzającego pizzę robota o nazwie bob, dziedziczącego atrybuty po trzech klasach — PizzaRobot, Chef oraz Employee. Wyświetlenie obiektu instancji bob wykonuje metodę `Employee.__repr__`, a danie robotowi podwyżki wywołuje metodę `Employee.giveRaise`, ponieważ mechanizm dziedziczenia odnajdzie tę metodę właśnie w tej klasie.

```
C:\code> python employees.py
<Employee: name=robert, salary=50000>
```

```
robert przygotowuje pizzę
<Employee: name=robert, salary=60000.0>
Employee robi różne rzeczy
Chef przygotowuje jedzenie
Server obsługuje klienta
PizzaRobot przygotowuje pizzę
```

W takiej hierarchii klas zazwyczaj możemy tworzyć instancje dowolnej klasy, nie tylko tej znajdującej się na dole. Pętla `for` w kodzie testującym moduł tworzy instancje wszystkich czterech klas. Każda z nich odpowiada w inny sposób, kiedy wywoła się metodę `work`, ponieważ w każdej klasie metoda ta jest inna. Na przykład robot `bob` uzyskuje metodę `work` z najbardziej specyficznej klasy (najniższej w hierarchii) `PizzaRobot`.

Tak naprawdę klasy *symulują* jedynie obiekty z prawdziwego świata. Metoda `work` wyświetla na razie tylko komunikat, jednak można ją rozszerzyć, tak by naprawdę wykonywała jakąś pracę (jeżeli dosłownie interpretujesz treść tego podrozdziału, zapoznaj się z interfejsami Pythona do takich urządzeń jak porty szeregowe, płytka Arduino lub Raspberry Pi).

Programowanie zorientowane obiektowo i kompozycja — związki typu „ma”

Pojęcie kompozycji zostało wprowadzone w rozdziałach 26. i 28. Z punktu widzenia programisty kompozycja obejmuje osadzanie innych obiektów w obiekcie pojemnika i aktywację ich w taki sposób, by implementowały jego metody. Dla projektanta kompozycja to inny sposób reprezentowania związków w określonej dziedzinie zastosowania. Zamiast jednak określać przynależność, kompozycja wiąże się z komponentami — częściami całości.

Kompozycja odzwierciedla również związki pomiędzy częściami, zazwyczaj nazywane *związkami typu „ma”* (ang. „has-a”). Niektóre teksty poświęcone projektowaniu w dziedzinie programowania zorientowanego obiektowo określają kompozycję jako *agregację* (lub rozróżniają te dwa pojęcia, wykorzystując agregację do opisania słabszych zależności pomiędzy pojemnikiem a komponentem). W niniejszej książce pojęcie „kompozycja” odnosi się po prostu do zbioru osadzonych obiektów. Klasa kompozytowa udostępnia własny interfejs i implementuje go, kierując osadzonymi obiektami.

Skoro już mamy pracowników, umieścimy ich w naszej pizzerii i czymś zajmiemy. Nasza pizzeria jest obiektem kompozytowym — ma piec, a także pracowników, takich jak kucharze czy osoby obsługujące klientów. Kiedy klient wchodzi do lokalu i składa zamówienie, komponenty restauracji zaczynają działać — kelner przyjmuje zamówienie, kucharz wytwarza pizzę i tak dalej. Poniższy przykład o nazwie `pizzashop.py` (działający w wersjach Pythona 2.x i 3.x) symuluje wszystkie obiekty oraz związki w tym scenariuszu.

```
# Plik pizzashop.py (wersja 2.x i 3.x)
from __future__ import print_function
from employees import PizzaRobot, Server
class Customer:
    def __init__(self, name):
        self.name = name
```

```

def order(self, server):
    print(self.name, "zamawia od", server)
def pay(self, server):
    print(self.name, "płaci za zamówienie", server)

class Oven:
    def bake(self):
        print("piec piecze")

class PizzaShop:
    def __init__(self):
        self.server = Server('Ernest')           # Osadzenie innych obiektów
        self.chef = PizzaRobot('Robert')         # Robot o imieniu Robert
        self.oven = Oven()

    def order(self, name):
        customer = Customer(name)             # Aktywacja innych obiektów
        customer.order(self.server)           # Klient zamawia od kelnera
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

    if __name__ == "__main__":
        scene = PizzaShop()                  # Utworzenie kompozytu
        scene.order('Amadeusz')              # Symulacja zamówienia
        Amadeusza
        print('...')
        scene.order('Aleksander')           # Symulacja zamówienia
        Aleksandra

```

Klasa `PizzaShop` jest pojemnikiem oraz kontrolerem. Jej konstruktor tworzy i osadza instancje klas pracowników, które utworzyliśmy przed chwilą, a także zdefiniowaną tutaj nową klasę `Oven`. Kiedy kod testujący tego modułu wywołuje metodę `order` klasy `PizzaShop`, osadzone obiekty są proszone o wykonanie swoich zadań jeden po drugim. Warto zwrócić uwagę na tworzenie nowego obiektu klasy `Customer` dla każdego zamówienia, a także przekazanie osadzonego obiektu `Server` do metod klasy `Customer`. Klienci pojawiają się i odchodzą, natomiast osoby ich obsługujące są częścią kompozytu pizzerii. Widać również, że pracownicy nadal znajdują się w relacji dziedziczenia — kompozycje oraz dziedziczenie są uzupełniającymi się narzędziami.

Kiedy wykonujemy ten moduł, nasza pizzeria obsługuje dwa zlecenia — jedno od Amadeusza, a drugie od Aleksandra.

C:\code> **python pizzashop.py**

Amadeusz zamawia od <Pracownik: imię=Ernest, wynagrodzenie=40000>

Robert przygotowuje pizzę

```
piec piecze
Amadeusz płaci za zamówienie <Pracownik: imię=Ernest, wynagrodzenie=40000>
...
Aleksander zamawia od <Pracownik: imię=Ernest, wynagrodzenie=40000>
Robert przygotowuje pizzę
piec piecze
Aleksander płaci za zamówienie <Pracownik: imię=Ernest, wynagrodzenie=40000>
```

Jest to oczywiście dziecinna symulacja, jednak obiekty oraz interakcje są reprezentatywne dla działania kompozytów. Klasy mogą reprezentować dowolne obiekty i związki, jakie jesteśmy w stanie opisać słownie. Wystarczy zastąpić *rzeczowniki* klasami, a *czasowniki* metodami i mamy już pierwszy szkic projektu.

Raz jeszcze procesor strumienia danych

By zobaczyć nieco bardziej realistyczny przykład, warto powrócić do funkcji uniwersalnego procesora strumienia danych, utworzonej w części we wprowadzeniu do programowania zorientowanego obiektowo w rozdziale 26.

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Zamiast korzystać tutaj z prostej funkcji, można również zapisać to w postaci klasy wykorzystującej kompozycję w celu uzyskania struktury i obsługiwanego dziedziczenia. W poniższym pliku *streams.py* (działającym w wersjach Pythona 2.x i 3.x) zaprezentowano sposób umieszczenia tego w kodzie klasy (zmieniona jest nazwa jednej z metod, ponieważ jest to plik przeznaczony do uruchomienia):

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
    def process(self):
        while True:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)
    def converter(self, data):
```

```
    assert False, 'konwerter musi być zdefiniowany' # Lub zgłoszenie
wyjątku
```

Ta klasa definiuje metodę converter, która musi być przeciążona w klasie potomnej. Jest to przykład modelu *klasy abstrakcyjnej*, szczegóły tej techniki można znaleźć w rozdziale 29. (część VII zawiera więcej informacji o instrukcji assert zgłaszającej wyjątek, gdy test zakończy się niepowodzeniem). Zapisane w ten sposób obiekty reader oraz writer osadzone są wewnętrz instancji klasy (*kompozycja*); logikę konwertera podajemy w klasie podzielonej, zamiast przekazywać funkcję konwertera (*dziedziczenie*). Plik *converters.py* pokazuje, jak można to zrobić.

```
from streams import Processor

class Uppercase(Processor):

    def converter(self, data):
        return data.upper()

    if __name__ == '__main__':
        import sys
        obj = Uppercase(open('spam.txt'), sys.stdout)
        obj.process()
```

Klasa Uppercase dziedziczy logikę pętli przetwarzającej strumień (i wszystko, co może być zapisane w kodzie jej klas nadzędnych). Musi zdefiniować tylko jeden unikalny element — logikę konwersji danych. Kiedy plik ten jest wykonywany, tworzy oraz wykonuje instancję wczytującą plik *trispam.txt* i wypisującą odpowiednik tego pliku zapisany wielkimi literami do strumienia wyjścia *stdout*.

```
C:\code> type trispam.txt
mielonka
Mielonka
MIELONKA!
C:\code> python converters.py
MIELONKA
MIELONKA
MIELONKA!
```

By przetwarzać inne rodzaje strumieni, należy przekazać inne rodzaje obiektów do wywołania konstruującego klasę. Poniżej zamiast strumienia wykorzystano plik wyjściowy.

```
C:\code> python
>>> import converters
>>> prog = converters.Uppercase(open('trispam.txt'), open('trispamup.txt',
'w'))
>>> prog.process()
C:\code> type trispamup.txt
MIELONKA
MIELONKA
```

MIELONKA!

Jak jednak zasugerowano wcześniej, moglibyśmy również przekazać dowolne obiekty opakowane w klasę, definiującą wymagane interfejsy metod wejścia oraz wyjścia. Poniżej znajduje się prosty przykład przekazujący klasę zapisującą opakowującą tekst w znaczniki języka HTML.

```
C:\code> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print('<PRE>%s</PRE>' % line.rstrip())
...
>>> Uppercase(open('spam.txt'), HTMLize()).process()
<PRE>MIELONKA</PRE>
<PRE>MIELONKA</PRE>
<PRE>MIELONKA!</PRE>
```

Śledząc przebieg sterowania w tym przykładzie, zobaczymy, że otrzymujemy *zarówno* konwersję tekstu na wielkie litery (dzięki dziedziczeniu), jak i formatowanie HTML (przez kompozycję), mimo że logika przetwarzania w oryginalnej klasie nadzędnej Processor nie wie nic o żadnym z tych kroków. Dla kodu przetwarzającego znaczenie ma jedynie to, by obiekty zapisujące miały metodę write, a także by została zdefiniowana metoda o nazwie convert. Nieistotne jest to, co te wywołania tak naprawdę robią. Taki polimorfizm oraz hermetyzacja logiki są podstawą dużego potencjału klas.

W tej chwili klasa nadzędna Processor udostępnia jedynie logikę przeglądającą plik. W bardziej realnym przykładzie moglibyśmy ją rozszerzyć w taki sposób, by obsługiwała dodatkowe narzędzia programistyczne dla klas podrzędnych i tym samym mogła stać się pełnowymiarową platformą. Utworzenie takiego narzędzia raz w klasie nadzędnej pozwala nam je zastosować ponownie we wszystkich naszych programach. Nawet w tym prostym przykładzie, ponieważ tak wiele zostało spakowane i odziedziczone w klasach, wystarczyło utworzyć kod z formatowaniem HTML. Całą resztę otrzymaliśmy gratis.

Inny przykład działania kompozycji można znaleźć w ćwiczeniu 9. pod koniec rozdziału 32. oraz w jego rozwiążaniu w części „Klasy i programowanie zorientowane obiektowo” w dodatku D — jest on podobny do przykładu z pizzerią. W książce skupiliśmy się na dziedziczeniu, ponieważ jest to podstawowe narzędzie programowania zorientowanego obiektowo w Pythonie. W praktyce jednak kompozycja wykorzystywana jest tak samo często jak dziedziczenie do strukturyzowania klas, w szczególności w większych systemach. Jak widzieliśmy, dziedziczenie oraz kompozycja często się dopełniają, a czasami są technikami alternatywnymi. Ponieważ kompozycja jest zagadnieniem z dziedziny projektowania pozostającym poza zakresem języka Python oraz niniejszej książki, osoby zainteresowane odsyłam po więcej informacji do innych źródeł.

Znaczenie klas oraz trwałości obiektów

W tej części książki kilka razy wspomniałem o serializacji za pomocą modułów pickle i shelve, ponieważ działa ona szczególnie dobrze w przypadku instancji klas. W rzeczywistości narzędzia do serializacji stanowią często wystarczająco atrakcyjną motywację do stosowania klas: zapisanie na dysku twardym obiektu klasy z użyciem

modułu `pickle` lub `shelve` jest prostą implementacją mechanizmu magazynowania danych oraz logiki aplikacji.

Poza symulowaniem interakcji ze świata rzeczywistego utworzone tutaj klasy pizzerii mogły również zostać wykorzystane jako podstawa trwałej bazy danych restauracji. Instancje klasy mogą być przechowywane na dysku w jednym kroku za pomocą modułów `pickle` lub `shelve` Pythona. Modułu `shelve` użyliśmy w rozdziale 28. do zapisywania na dysku obiektów klas, również interfejs serializacji obiektów za pomocą modułu `pickle` jest zadziwiająco prosty w użyciu.

```
import pickle

object = someClass()

file = open(filename, 'wb')                                # Utworzenie pliku
zewnętrznego

pickle.dump(object, file)                                 # Zapisanie obiektu w pliku

import pickle

file = open(filename, 'rb')

object = pickle.load(file)                               # Pobranie go z powrotem
później
```

Moduł `pickle` konwertuje obiekty znajdujące się w pamięci na zserializowane strumienie bajtów (ciągi znaków), które mogą być przechowywane w plikach czy przesyłane za pośrednictwem sieci. Przeciwna operacja konwertuje strumień bajtów z powrotem na identyczne obiekty w pamięci. Moduł `shelve` jest podobny, jednak automatycznie serializuje on obiekty na bazę danych dostępną po kluczu, eksportującą interfejs podobny do słownika.

```
import shelve

object = someClass()

dbase = shelve.open('filename')

dbase['key'] = object                                    # Zapisanie pod kluczem

import shelve

dbase = shelve.open('filename')

object = dbase['key']                                  # Pobranie z powrotem później
```

W naszym przykładzie wykorzystanie klas do modelowania pracowników oznacza, że możemytrzymać prostą bazę danych pracowników i firm za pomocą niewielkiej ilości pracy. Serializacja takich obiektów instancji do pliku sprawia, że są one trwałe pomiędzy niezależnymi wywołaniami programów w Pythonie.

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Pracownik: imię=Ernest, wynagrodzenie=40000>, <Pracownik: imię=Robert,
wynagrodzenie=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.dat', 'wb'))
```

Ta prosta operacja skutkuje zapisem do pliku na dysku całej skomplikowanej struktury obiektu shop. Taki magazyn danych może być odczytany w innej sesji tego samego programu, a nawet w zupełnie osobnym programie. Obiekty zapisane w ten sposób przechowują swój stan i zachowanie.

```
>>> import pickle  
>>> obj = pickle.load(open('shopfile.dat', 'rb'))  
>>> obj.server, obj.chef  
(<Pracownik: imię=Ernest, wynagrodzenie=40000>, <Pracownik: imię=Bob,  
wynagrodzenie=50000>)  
>>> obj.order('Zuzanna')  
Zuzanna zamawia od <Pracownik: imię=Ernest, wynagrodzenie=40000>  
Robert przygotowuje pizzę  
piec piecze  
Zuzanna płaci za zamówienie <Pracownik: imię=Ernest, wynagrodzenie=40000>
```

Powyższa symulacja jest bardzo prosta, ale można rozbudować klasę shop tak, aby rejestrowała zapasy, przychody itp. i zapisywała dane do pliku. Dzięki temu stan klasy może być zachowywany po wprowadzeniu w nim zmian. Więcej przykładów użycia modułów pickle i shelfe można znaleźć w podręczniku standardowej biblioteki Pythona oraz w rozdziałach 9., 28. i 37.

Programowanie zorientowane obiektowo a delegacja — obiekty „opakowujące”

Zwolennicy programowania zorientowanego obiektowo często wspominają o *delegacji*, co zazwyczaj dotyczy obiektów kontrolerów osadzających inne obiekty, do których przekazują żądania operacji. Kontrolery mogą się zajmować zadaniami administracyjnymi, takimi jak na przykład śledzenie i weryfikowanie dostępu, wprowadzanie dodatkowych operacji do komponentów interfejsu czy kontrolowanie aktywnych instancji.

Można powiedzieć, że delegacja jest specjalną formą kompozycji z jednym osadzonym obiektem zarządzanym przez klasę *opakowującą* (czasami nazywaną *klasą pośredniczącą*, ang. *wrapper* lub *proxy class*) zachowującą większość lub całość interfejsu osadzonego obiektu. Pojęcie klasy pośredniczącej odnosi się również do innych mechanizmów, na przykład funkcji. W delegacji mamy do czynienia z pośrednictwem wszystkich działań obiektu, w tym wywoływaniem metod i wykonywaniem innych operacji.

Pojęcie delegacji zostało zaprezentowane na przykładzie 28. W Pythonie jest ona często implementowana za pomocą metody `__getattribute__`, którą poznaliśmy w rozdziale 30. Ponieważ przechytuje ona dostęp do nieistniejących atrybutów, klasa opakowująca może wykorzystać metodę `__getattribute__` do przekierowania dowolnego dostępu do opakowanego obiektu. Ponieważ metoda ta może generycznie kierować żdaniami odczytu atrybutów, klasa pośrednicząca zachowuje interfejs opakowanego obiektu i może dodawać własne operacje.

Rozważmy na przykład poniższy plik *trace.py* (działający tak samo w wersjach 2.x i 3.x).

```
class wrapper:  
    def __init__(self, object):
```

```

    self.wrapped = object                         # Zapisanie obiektu

def __getattr__(self, attrname):
    print('Śledzenie: ' + attrname)             # Śledzenie pobrania
    return getattr(self.wrapped, attrname)        # Delegacja pobrania

```

Łatwo sobie przypomnieć, że zgodnie z informacjami z rozdziału 30. metoda `__getattr__` pobiera nazwę atrybutu w postaci łańcucha znaków. Kod ten wykorzystuje wbudowaną funkcję `getattr` do pobrania atrybutu z opakowanego obiektu po łańcuchu znaków jego nazwy — `getattr(X, N)` jest tym samym co `X.N`, oprócz tego, że `N` jest wyrażeniem, które w czasie wykonywania obliczane jest do łańcucha znaków, a nie do zmiennej. Tak naprawdę wywołanie `getattr(X, N)` jest podobne do `X.__dict__[N]`, jednak ta pierwsza wersja wykonuje również wyszukiwanie dziedziczenia, podobnie jak `X.N`, podczas gdy ta druga tego nie robi (więcej informacji na temat atrybutu `__dict__` można znaleźć w rozdziałach 22. i 29.).

Takie podejście do klas opakowujących modułu można wykorzystać do zarządzania dostępem do dowolnego obiektu z atrybutami — list, słowników, a nawet klas oraz instancji. W poniższym przykładzie klasa `wrapper` po prostu wyświetla komunikat śledzenia dla każdego dostępu do atrybutu i deleguje żądanie atrybutu do opakowanego obiektu `wrapped`.

```

>>> from trace import wrapper
>>> x = wrapper([1,2,3])                                # Opakowanie listy
>>> x.append(4)                                         # Delegacja do metody listy
Śledzenie: append
>>> x.wrapped                                         # Wyświetlenie mojej składowej
[1, 2, 3, 4]
>>> x = wrapper({"a": 1, "b": 2})                      # Opakowanie słownika
>>> x.keys()                                           # Delegacja do metody słownika
Śledzenie: keys
['a', 'b']

```

Rezultatem jest rozszerzenie całego interfejsu opakowanego obiektu za pomocą dodatkowego kodu w klasie `Wrapper`. Takie rozwiązanie można wykorzystać na przykład do logowania wywołań metod albo przekierowywania wywołań metod do dodatkowej czy specjalnej logiki, dostosowania klasy do nowego interfejsu itp.

Do pojęć obiektów opakowanych oraz delegowanych operacji powróćmy w następnym rozdziale jako do jednego ze sposobów rozszerzania typów wbudowanych. Osoby zainteresowane wzorcem projektowym delegacji powinny zapoznać się z rozdziałami 32. i 39., w których zostały opisane *dekoratory funkcji* — pokrewne pojęcie rozszerzające wybrane funkcje lub metody, a nie cały interfejs obiektu — oraz *dekoratory klas* służące do automatycznego dodawania klas opakowujących do wszystkich instancji.



Uwaga na temat zgodności: jak widzieliśmy w przykładzie w rozdziale 28., w wersji 3.x delegowanie interfejsów obiektów przez ogólne *klasy pośredniczące* wygląda zupełnie inaczej, gdy obiekty opakowujące implementują metody przeciążające operatory. Z technicznego punktu widzenia jest to nowość wprowadzona przez klasy w nowym stylu, która może pojawiać się również w kodzie w wersji 2.x. Jak się przekonamy w następnym rozdziale, jest to obowiązkowa zmiana, często skutkująca przejściem do wersji 3.x.

W Pythonie 2.x metody przeciążające operatory wywoływane w kontekście wbudowanych operatorów są wywoływanie za pośrednictwem ogólnych mechanizmów uzyskiwania dostępu do atrybutów, jak metoda `__getattribute__`. Na przykład bezpośrednie wyświetlenie opakowanego obiektu za pomocą metody `__repr__` lub `__str__` spowoduje wywołanie metody `__getattribute__`, a następnie przekazanie wywołania do opakowanego obiektu. Podobna zasada dotyczy metod `__iter__`, `__add__` i metod innych operatorów opisanych w poprzednim rozdziale.

W Pythonie 3.x taka sytuacja nie ma już miejsca: wyświetlenie obiektu nie wywołuje metody `__getattribute__` (ani jej krewnej `__getattribute__`, którą poznamy w następnym rozdziale), przez co wykorzystywana jest domyślna metoda reprezentacji tekstowej obiektu. W wersji 3.x klasy w nowym stylu wyszukują metody niejawnie wywoływane przez wbudowane operacje w klasach i całkowicie pomijają zwykłe instancje. Jawne odwołania do atrybutów są kierowane do metody `__getattribute__` w taki sam sposób w wersjach 2.x i 3.x, jednak wyszukiwanie metod operacji wbudowanych różni się sposobem, który może mieć wpływ na niektóre narzędzia oparte na delegacji.

Do tego zagadnienia wróćmy w następnym rozdziale poświęconym klasom w nowym stylu oraz w rozdziałach 38. i 39. w kontekście atrybutów zarządzanych i dekoratorów. W tej chwili należy jedynie pamiętać, że podczas stosowania wzorca kodowania delegacji może być konieczne ponowne zdefiniowanie metod przeciążających operatory w klasach opakowujących (ręcznie albo za pomocą narzędzi lub klas nadzorzących), jeśli będą używane przez osadzone obiekty i mają być przechwytywane przez klasy w nowym stylu.

Pseudoprыватне атрибути клас

Obok zagadnień strukturalnych dobry projekt zorientowany obiektowo powinien uwzględniać kwestie użycia nazw. W przykładzie w rozdziale 28. dowiedzieliśmy się, że metody zdefiniowane w ogólnej klasie mogą być modyfikowane w klasach podrzędnych (jeżeli zostaną wyeksponowane). Poznaliśmy też związane z tym podejściem kompromisy, które umożliwiają wprawdzie dostosowywanie metod i bezpośrednich wywołań, ale również narażają je na przypadkowe zastąpienia.

W części V dowiedzieliśmy się, że każda zmienna przypisana na najwyższym poziomie pliku modułu jest eksportowana. Domyślnie tak samo jest również w przypadku klas — ukrywanie danych jest konwencją, a klient może pobrać lub zmodyfikować każdy atrybut klasy lub instancji w taki sposób, na jaki ma ochotę. Tak naprawdę wszystkie atrybuty są — zgodnie z terminologią języka C++ — „publiczne” oraz „wirtualne”. Są dostępne z każdego miejsca i wyszukiwane dynamicznie w czasie wykonywania [1].

To wszystko nadal jest prawdziwe. Python obsługuje jednak również pojęcie „znieksztalconia” nazw zmiennych (to jest rozszerzania) w celu lokalizacji niektórych zmiennych w klasach. Znieksztalcone nazwy są czasami zwodniczo nazywane „atributami prywatnymi”, jednak tak naprawdę jest to jedynie sposób *lokalizacji* zmiennej do klasy, która ją utworzyła — znieksztalconie nie zapobiega dostępowi z kodu znajdującego się poza klasą. Opcja ta ma w zamierzeniach zapobiec konfliktom przestrzeni nazw w instancjach, a nie służyć do ogólnego ograniczenia dostępu do zmiennych. Znieksztalcone zmienne lepiej jest zatem nazywać „pseudoprыватnymi” niż „prywatkymi”.

Zmienne pseudoprыватne są zaawansowane i całkowicie opcjonalne. Prawdopodobnie nie przydadzą nam się do niczego, dopóki nie zaczniemy pisać dużych hierarchii klas w projektach tworzonych przez wielu programistów. W rzeczywistości nie zawsze są używane nawet tam, gdzie w teorii powinny. Programiści Pythona wewnętrzne nazwy w klasach najczęściej

sygnalizują z użyciem przedrostka złożonego z pojedynczego podkreśnika (`_X`), co stanowi nieformalną konwencję nazewnictwa mającą sugerować, że mamy do czynienia z nazwą, której nie należy modyfikować (natomiast dla Pythona ta konwencja nie ma żadnego znaczenia).

Ponieważ jednak możemy spotkać się z tą opcją w kodzie utworzonym przez kogoś innego, musimy być przynajmniej świadomie jej działania, nawet jeśli sami nie będziemy z niej korzystać. Gdy poznamy ich zalety i konteksty użycia, okaże się, że są bardziej przydatne, niż to sobie wyobrażaliśmy.

Przegląd znieksztalcania nazw zmiennych

Poniżej widać, jak działa znieksztalconie. Nazwy zmiennych wewnętrz instrukcji `class` zaczynające się od podwójnych znaków `_`, ale nie kończące się takimi samymi znakami, są automatycznie rozszerzane w taki sposób, by zawierały również nazwę zawierającą je klasy. Na przykład zmienna `_X` wewnętrz klasy o nazwie `Spam` zostaje automatycznie zmieniona na `_Spam__X` — oryginalną nazwę poprzedza się pojedynczym znakiem `_` oraz nazwą klasy zawierającej zmienną. Ponieważ zmodyfikowana nazwa zawiera nazwę klasy, jest w pewnym stopniu unikalna. Nie będzie wchodziła w konflikt z podobnymi nazwami utworzonymi przez inne klasy hierarchii.

Znieksztalconie nazw ma miejsce jedynie w instrukcji `class` i tylko dla zmiennych, których nazwy rozpoczynają się od dwóch znaków `_`. Ma jednak miejsce dla *każdej* nazwy poprzedzonej dwoma takimi znakami, w tym atrybutów klas (również nazw metod) i atrybutów instancji przypisanych do `self`. Na przykład w klasie `Spam` referencja do atrybutu instancji `self._X` została przełożona na `self._Spam__X`, a referencja do atrybutu instancji `self.__X` byłaby przekształcona na `self._Spam__X`.

Pomimo znieksztalcen wszystkie referencje funkcjonują poprawnie, o ile podwójne znaki podkreślenia są stosowane wszędzie tam, gdzie klasa odwołuje się do danej nazwy. Ponieważ dodawać atrybuty do instancji może więcej niż jedna klasa, znieksztalconie pomaga zapobiec konfliktom. Żeby jednak zobaczyć, jak się to odbywa, musimy przejść do przykładu.

Po co używa się atrybutów pseudoprzywatnych

Jednym z podstawowych problemów, które ma rozwiązać stosowanie nazw pseudoprzywatnych, jest sposób zapisu atrybutów instancji klas. W Pythonie wszystkie atrybuty instancji są przyłączone do jego obiektu na samym dole drzewa dziedziczenia i są współdzielone przez wszystkie metody klasy, którym została przekazana instancja. To różni Pythona od modelu stosowanego w C++, w którym każda klasa otrzymuje własną przestrzeń dla zdefiniowanych w niej atrybutów (danych).

Wewnętrz metody klasy w Pythonie za każdym razem, gdy metoda przypisuje coś do atrybutu `self` (na przykład `self.atrybut = wartość`), modyfikuje lub tworzy atrybut w instancji (wyszukiwanie dziedziczenia odbywa się jedynie przy referencjach, nie przy przypisaniu). Ponieważ jest tak nawet wtedy, gdy wiele klas z hierarchii przypisuje wartość do tego samego atrybutu, konflikty są możliwe.

Załóżmy na przykład, że kiedy programista tworzy klasę, zakłada, że ma atrybut o nazwie `X` w instancji. W metodach tej klasy nazwa ta zostaje ustawniona, a później pobrana.

```
class C1:  
    def meth1(self): self.x = 88 # Zakładam, że X jest moje  
    def meth2(self): print (self.x)
```

Załóżmy jeszcze, że inny programista, pracujący w odosobnieniu, poczyni to samo założenie w tworzonej przez siebie klasie.

```

class C2:
    def metha(self): self.x = 99          # Ja też tak uważam
    def methb(self): print (self.x)

```

Obie klasy same z siebie działają dobrze. Problem pojawia się, kiedy zostaną one zmieszane ze sobą w tym samym drzewie klas.

```

class C3(C1, C2): ...
I = C3()                      # Tylko jedna zmienna X w I!

```

Teraz wartość zwracana przez każdą z klas dla kodu `self.x` będzie uzależniona od tego, która klasa przypisała ją jako ostatnia. Ponieważ wszystkie przypisania do `self.x` odnoszą się do jednego instancji, istnieje tylko jeden atrybut `X` — `I.x` — bez względu na to, ile klas wykorzystuje tę samą nazwę atrybutu.

Nie jest to problem, jeśli taki jest oczekiwany efekt. Tak właśnie komunikują się klasy — instancja jest pamięcią współdzieloną. By zagwarantować, że atrybut należy do klasy, która go wykorzystuje, należy poprzedzić jego nazwę podwójnymi znakami `_` w każdym miejscu, w jakim jest użyty w klasie, tak jak w poniższym pliku `pseudoprivate.py` (działa w wersjach 2.x i 3.x).

```

class C1:
    def meth1(self): self._X = 88          # Teraz X jest moje
    def meth2(self): print(self._X)        # Staje się _C1_X w I
class C2:
    def metha(self): self._X = 99          # Jest też moje
    def methb(self): print(self._X)        # Staje się _C2_X w I
class C3(C1, C2): pass
I = C3()                          # Dwie zmienne X w I
I.meth1(); I.metha()
print I.__dict__
I.meth2(); I.methb()

```

Dzięki takiemu przedrostkowi atrybuty `X` zostaną przed dodaniem do instancji rozszerzone w taki sposób, by obejmować również nazwy klas. Jeśli wykonamy wywołanie `dir` dla `I` lub przejrzymy słownik przestrzeni nazw tej instancji po przypisaniu atrybutów, zobaczymy w nim rozszerzone zmienne `_C1_X` oraz `_C2_X`, ale już nie samo `X`. Ponieważ takie rozszerzenie sprawia, że nazwy te stają się unikalne wewnątrz instancji, osoby tworzące klasy mogą zakładać, że są prawdziwymi właścicielami wszystkich zmiennych poprzedzonych dwoma znakami `_`.

```

C:\code> python pseudoprivate.py
{'_C2_X': 99, '_C1_X': 88}
88
99

```

Powyższa sztuczka pozwala uniknąć potencjalnych konfliktów nazw w instancji, jednak należy pamiętać, że nie jest to jednoznaczne z prawdziwą prywatnością zmiennych. Jeśli znamy nazwę klasы zawierającej zmienną, nadal możemy uzyskać dostęp do atrybutów w dowolnym miejscu, w którym mamy referencję do instancji, wykorzystując rozszerzoną nazwę zmiennej (na

przykład I._C1_X = 77). Co więcej, konflikt nazw i tak może się pojawić, jeżeli nieświadomy programista będzie jawnie używał rozwiniętych nazw (jest to mało prawdopodobny, jednak możliwy przypadek). Z drugiej strony opcja ta sprawia, że o wiele mniej prawdopodobne jest przypadkowe natknęcie się na zmienne klas.

Atrybuty pseudoprivałte przydają się również w większych bibliotekach lub narzędziach. Dzięki nim unika się wprowadzania nowych nazw metod, które zmniejszą prawdopodobieństwo zastąpienia wewnętrznych metod nazwami zdefiniowanymi niżej w drzewie, jak również mogą przypadkowo przesłaniać definicje stosowane w innym miejscu drzewa. Jeśli dana metoda ma być używana wyłącznie w jednej klasie, która może być osadzana w innych klasach, nazwa poprzedzona dwoma znakami podkreślenia nie będzie kolidowała z innymi nazwami w drzewie dziedziczenia, co jest szczególnie użyteczne w przypadku dziedziczenia wielokrotnego.

```
class Super:  
    def method(self): ... # Zwykła metoda  
  
class Tool:  
    def __method(self): ... # Nazwa zostanie niejawnie  
    zmieniona na _Tool__method  
    def other(self): self.__method() # Użycie metody wewnętrznej  
  
class Sub1(Tool, Super): ...  
    def actions(self): self.method() # Wywołuje Super.method()  
  
class Sub2(Tool):  
    def __init__(self): self.method = 99 # Nie psuje Tool.__method
```

Z wielokrotnym dziedziczeniem spotkaliśmy się już w rozdziale 26., a w dalszej części tego rozdziału pogłębimy to zagadnienie. Jak pamiętamy, klasy nadrzędne są przeszukiwane w kolejności od lewej do prawej. W powyższym przykładzie oznacza to, że obiekty klasy Sub1 będą preferowały atrybuty zdefiniowane w klasie Tool — przed atrybutami zdefiniowanymi w klasie Super. W tym przykładzie moglibyśmy wpłynąć na tę kolejność, zamieniając w deklaracji kolejność klas nadrzędnych, ale dzięki metodom pseudoprivałtym mamy jeszcze większą kontrolę. Nazwy pseudoprivałte zabezpieczają przed przypadkowym przesłonięciem nazwy, co demonstruje definicja klasy Sub2.

Podkreślę jeszcze raz, że ta cecha języka powstała z myślą o dużych projektach, w których bierze udział wielu programistów. Jednak również w takich przypadkach nie należy przesadzać. Nie warto niepotrzebnie zaśmiecać kodu. Mechanizm ten powinien być stosowany wyłącznie w celu zapewnienia kontroli nad wybranymi nazwami przez określoną klasę. W przypadku prostych programów użycie tej konstrukcji najczęściej mija się z celem.

Więcej przykładów zastosowania nazw atrybutów w konwencji __X można znaleźć w dalszej części rozdziału przy okazji omawiania modułu *lister.py*, w punkcie poświęconym wielokrotnemu dziedziczeniu oraz w punkcie omawiającym dekorator klas **Private** w rozdziale 39.

Jeśli ktoś jest zainteresowany emulacją prywatnych atrybutów instancji, szkic takiego rozwiązania znajdzie w punkcie „Metody `__getattribute__` oraz `__setattr__` przechwytyują referencje do atrybutów” w rozdziale 30. oraz w opartym na tej koncepcji wspomnianym dekoratorze klas **Private** w rozdziale 39. W klasach Pythona mamy możliwość pełnej kontroli nad dostępem do atrybutów, jednak takie podejście jest praktykowane niezwykle rzadko, nawet w bardzo dużych systemach.

Metody są obiektami — z wiązaniem i bez wiązania

Metody, a w szczególności metody związane, upraszczają implementację wielu wzorców projektowych w Pythonie. Metody związane omawialiśmy w skrócie w rozdziale 30., przy okazji omawiania metody `__call__`. W niniejszym rozdziale omówimy ten temat bardziej szczegółowo i przekonamy się, że to mechanizm znacznie bardziej ogólny i elastyczniejszy, niż mogłoby się wydawać.

W rozdziale 19. dowiedzieliśmy się, w jaki sposób przetwarzają funkcje, traktując je jak zwykłe obiekty. Również metody są obiektami i można ich używać podobnie jak innych obiektów: mogą być przypisywane do zmiennych, umieszczane w argumentach funkcji, zapisywane w strukturach danych itp. Podobnie jak proste funkcje, metody traktowane są jako obiekty „pierwszej klasy”. Dostęp do metod klas uzyskuje się z instancji lub klasy i dle tego w Pythonie istnieją dwa rodzaje metod.

Obiekty metod klas bez wiązania (ang. unbound) — bez self

Dostęp do atrybutu funkcji klasy za pomocą składni kwalifikującej z klasą zwraca obiekt metody bez wiązania. By wywołać metodę, musimy w sposób jawnym dostarczyć jej w pierwszym argumencie obiekt instancji. W Pythonie 3.x metoda niezwiązana jest traktowana tak samo jak zwykła funkcja i może być wywołana z poziomu klasy. W 2.x metody niezwiązane są osobnym typem i nie można ich wywoływać bez podania instancji.

Obiekty metod instancji z wiązaniem (ang. bound) — pary self + funkcja

Dostęp do atrybutu funkcji klasy za pomocą składni kwalifikującej z *instancją* zwraca obiekt metody z wiązaniem. Python automatycznie pakuje instancję z funkcją w obiekt metody z wiązaniem, dzięki czemu nie musimy do wywołania tej metody przekazywać instancji.

Oba rodzaje metod są pełnowymiarowymi obiektami — można je dowolnie przemieszczać w ramach programu, podobnie jak ciągi znaków lub listy. Oba wymagają również przy wykonywaniu podania w pierwszym argumencie instancji (to znaczy wartości dla `self`). Dlatego właśnie musielibyśmy przekazać instancję w sposób jawnym, kiedy wywoływaliśmy metody klas nadzędnych z metod klas podrzędnych w opisanych przykładach (również w wykorzystanym w tym rozdziale pliku *employees.py*). Z technicznego punktu widzenia wywołania takie tworzą obiekty metod bez wiązania.

Kiedy wywołuje się obiekt metody z wiązaniem, Python automatycznie udostępnia nam instancję — wykorzystywaną do tworzenia obiektu metody z wiązaniem. Oznacza to, że obiekty metod z wiązaniem są zazwyczaj wymienne z prostymi obiektami funkcji, co czyni je szczególnie przydatnymi dla interfejsów oryginalnie utworzonych dla funkcji (realistyczny przykład można znaleźć w ramce „Znaczenie metod z wiązaniem oraz wywołań zwrotnych”).

By to zilustrować, założymy, że definiujemy następującą klasę.

```
class Spam:  
    def doit(self, message):  
        print message
```

Teraz w normalnej operacji tworzymy instancję i wywołujemy jej metodę w celu wyświetlenia przekazanego argumentu.

```
object1 = Spam()  
object1.doit('Witaj, świecie!')
```

Tak naprawdę jednak po drodze, tuż przed nawiasami wywołania metody, generowany jest obiekt metody z wiązaniem. Możemy nawet pobrać metodę z wiązaniem bez prawdziwego wywoływanego jej. Składnia kwalifikująca `obiekt.atrybut` jest wyrażeniem obiektu. W poniższym kodzie zwraca ona obiekt metody z wiązaniem, pakujący instancję (`object1`) z metodą funkcji (`Spam.doit`). Możemy przypisać tę metodę z wiązaniem do innej nazwy, a następnie wywołać ją tak, jakby była prostą funkcją.

```
object1 = Spam()
x = object1.doit
instancia + funkcja
# Obiekt metody z wiązaniem:

x('Witaj, świecie!')
object1.doit('...')

# Ten sam efekt co
```

Z drugiej strony, jeśli użyjemy składni kwalifikującej z klasą w celu dotarcia do metody `doit`, z powrotem otrzymamy obiekt metody bez wiązania, będący po prostu referencją do obiektu funkcji. By wywołać metodę tego typu, musimy przekazać instancję jako argument znajdujący się najbardziej na lewo — w wyrażeniach nie jest stosowana, ale metoda jej wymaga:

```
object1 = Spam()
t = Spam.doit
t(object1, 'siema')
# Obiekt metody bez wiązania
# Przekazanie instancji
```

Przez rozszerzenie te same reguły mają zastosowanie wewnątrz metody klasy, jeśli odniesiemy się do atrybutów `self` odnoszących się do funkcji w klasie. Wyrażenie `self.metoda` jest obiektem metody z wiązaniem, ponieważ `self` jest obiektem instancji.

```
class Eggs:
    def m1(self, n):
        print(n)
    def m2(self):
        x = self.m1
        # Inny obiekt metody z
        wiązaniem
        x(42)
        # Wygląda jak prosta funkcja
Eggs().m2()
# Wyświetla 42
```

Zazwyczaj metody wywołuje się natychmiast po pobraniu ich za pomocą składni kwalifikującej, dlatego nie zawsze zauważamy obiekty wygenerowane po drodze. Jeśli jednak zaczniemy pisać kod wywołujący obiekty w sposób uniwersalny, musimy uważać na specjalne traktowanie metod bez wiązania — normalnie wymagają one jawnego przekazania obiektu instancji.



Opcjonalnym wyjątkiem od tej reguły są *metody statyczne i metody klasy* opisane w następnym rozdziale i krótko wspomniane w jednym z poniższych podrozdziałów. Metody statyczne, podobnie jak wiązane, mogą udawać zwykłe funkcje, ponieważ do ich wywołania nie są potrzebne instancje. Formalnie w Pythonie istnieją trzy rodzaje metod na poziomie klasy: metody instancji, metody statyczne i metody klasy. W wersji 3.x można w klasach stosować również zwykłe funkcje. W rozdziale 40. wyróżnione zostały dodatkowo metody metaklasy, które w rzeczywistości są metodami klasy o mniejszych zakresach.

W wersji 3.x metody niezwiązane są funkcjami

W Pythonie 3.x porzucono koncepcję metod *niezwiązanych*. To, co w niniejszym rozdziale nazywamy metodą niezwiązaną, jest w Pythonie 3.x traktowane tak samo jak funkcja. W większości przypadków nie powinno to mieć znaczenia w napisanym kodzie — w pierwszym parametrze wywołania takiej metody wywołanej na obiekcie otrzymamy obiekt klasy.

Jedyny problem będą stanowić programy, w których stosowana jest kontrola typów. W Pythonie 2.x wywołanie funkcji `type` na takiej metodzie klasy zwróci ciąg znaków `unbound method`, natomiast w Pythonie 3.x otrzymamy `function`.

Co więcej, w 3.x wszystkie metody można wywoływać bez obiektu (instancji klasy), jeśli metoda go nie potrzebuje, dzięki czemu można je wywoływać z poziomu klasy, nie obiektu. Innymi słowy, Python 3.x przekaże instancję do metody tylko w przypadku wywołania tej metody z poziomu instancji. W przypadku wywołania z poziomu klasy instancja (o ile jest potrzebna metodzie) musi być przekazana ręcznie.

```
C:\code> c:\python33\python
>>> class Selfless:
...     def __init__(self, data):
...         self.data = data
...     def selfless(arg1, arg2):          # W 3.x jest zwykłą funkcją
...         return arg1 + arg2
...     def normal(self, arg1, arg2):      # oczekuje instancji
...         return self.data + arg1 + arg2
...
>>> X = Selfless(2)
>>> X.normal(3, 4)                  # instancja jest przekazana
automatycznie
9
>>> Selfless.normal(X, 3, 4)        # parametr self oczekiwany przez
metodę przekazany ręcznie
9
>>> Selfless.selfless(3, 4)          # brak instancji: działa w 3.x,
ale nie w 2.x!
7
```

Ostatnie wywołanie z powyższego przykładu w Pythonie 2.x zakończy się błędem, ponieważ metody niezwiązane wymagają przekazania instancji klasy. W 3.x wywołanie to zadziała dzięki temu, że metody niezwiązane są traktowane jak funkcje i nie wymagają przekazania instancji. W ten sposób traci się w pewnym stopniu możliwość wychwytywania błędów w kodzie w wersji 3.x (na przykład gdy mylnie nie przekaże się instancji klasy), ale z drugiej strony daje to więcej swobody, pozwalając używać metody tak samo jak zwykłych funkcji, jeśli instancja klasy nie zostanie przekazana, a w metodzie nie jest wykorzystywany argument `self`.

Poniższe dwa wywołania nie zadziałają ani w 2.x, ani w 3.x. Pierwsze z nich (wywołane z obiektu) powoduje automatyczne przekazanie obiektu do metody, która go nie oczekuje. Drugie (wywołane z klasy) nie przekazuje klasy do metody, która jej oczekuje:

```
>>> X.selfless(3, 4)
```

```
TypeError: selfless() takes exactly 2 positional arguments (3 given)
>>> Selfless.normal(3, 4)
TypeError: normal() takes exactly 3 positional arguments (2 given)
```

Dzięki tej zmianie od wersji 3.x do definiowania metod nieobsługujących parametru `self` nie jest konieczne wykorzystywanie opisanego w następnym rozdziale dekoratora `staticmethod`. Tego typu metody są wywoływanie jak zwykłe funkcje, bez przekazywania obiektu w argumencie. W 2.x tego typu wywołania wywołują błąd, chyba że obiekt instancji będzie przekazany ręcznie. Więcej szczegółów na temat metod statycznych poznamy w następnym rozdziale.

Należy znać zmiany wprowadzone w zakresie metod niezwiązańych w Pythonie 3.x, ale metody te mają o wiele większe znaczenie praktyczne. Instancja i funkcja są łączone w jeden obiekt, który można traktować tak samo jak inne obiekty wywoływane (ang. *callable*). W kolejnym punkcie zademonstruję praktyczne zastosowania tej właściwości.



Aby lepiej zrozumieć sposoby wykorzystania metod niezwiązanych w Pythonie 3.x i 2.x, warto zajrzeć do przykładu *lister.py* w punkcie omawiającym wielokrotne dziedziczenie w dalszej części rozdziału. Przedstawiam tam implementację klas służących do wypisywania atrybutów instancji i klas, działających w obydwu wersjach Pythona — w wersji 2.x jako metody bez wiązania, a w 3.x jako proste funkcje. Należy też zwrócić uwagę, że jest to zmiana właściwa dla wersji 3.x, a nie dla obowiązkowego w niej modelu klas w nowym stylu.

Metody związane i inne obiekty wywoływane

Jak wspomniałem wcześniej, metody związane mogą być przetwarzane jako uogólnione obiekty, podobnie jak zwykłe funkcje. Innymi słowy, metody można przekazywać tak samo jak inne obiekty. Co więcej, dzięki temu, że metody związane łączą funkcję i instancję w jeden pakiet, mogą być wywoływane jak każdy inny obiekt wywoływany (*callable*) i nie wymagają stosowania specjalnej składni. W poniższym listingu definiujemy cztery obiekty metod związanych i wykorzystujemy je później tak, jak zwykłe wywołania funkcji.

```

4
>>> acts = [x.double, y.double, y.triple, z.double] # Lista metod związanych
>>> for act in acts:                                # Opóźnienie wywołań
...     print(act())                                # Wywołanie w trybie
zwykłych funkcji

...
4
6
9
8

```

Obiekty metod związanych oferują mechanizm introspekcji, podobnie jak zwykłe funkcje. Dzięki temu mamy dostęp do związanego obiektu instancji klasy oraz funkcji metody. Wywołanie obiektu metody związanej powoduje wywołanie tej funkcji na obiekcie instancji:

```

>>> bound = x.double
>>> bound.__self__, bound.__func__
(<__main__.Number object at 0x...itd... >, <function Number.double at 0x...
itd...>)
>>> bound.__self__.base
2
>>> bound()                                         # Wywołuje
bound.__func__(bound.__self__, ...)
4

```

Inne obiekty wywoływanie

Metody związane to w rzeczywistości jeden z przykładów użytecznych obiektów wywoływanych oferowanych przez Pythona. Poniższy listing ilustruje własności obiektów wywoływanych — funkcje definiowane z użyciem instrukcji `def`, `lambda`, jak również instancje klas definiujących metodę `__call__` obsługującą interfejs wywoływania i można je wykorzystywać w ten sam sposób:

```

>>> def square(arg):
...     return arg ** 2                                # Zwykłe funkcje (def lub
lambda)

...
>>> class Sum:
...     def __init__(self, val):                      # Klasa wywoływana
...         self.val = val
...     def __call__(self, arg):
...         return self.val + arg
...

```

```

>>> class Product:
...     def __init__(self, val):                      # Metody związane
...         self.val = val
...     def method(self, arg):
...         return self.val * arg
...
...
>>> sobject = Sum(2)
>>> pobject = Product(3)
>>> actions = [square, sobject, pobject.method] # Funkcja, instancja, metoda
>>> for act in actions:                         # wszystkie z nich wywołuje
się tak samo
...     print(act(5))                            # jako obiekt wywoływany z
jednym argumentem
...
25
7
15
>>> actions[-1](5)                           # Indeksowanie, składanie,
mapy
15
>>> [act(5) for act in actions]
[25, 7, 15]
>>> list(map(lambda act: act(5), actions))
[25, 7, 15]

```

Znaczenie metod z wiązaniem oraz wywołań zwrotnych

Ponieważ metody z wiązaniem automatycznie łączą instancje oraz funkcje metod klasy w pary, możemy je wykorzystywać we wszystkich miejscach, w których oczekiwana jest prosta funkcja. Jednym z najczęściej spotykanych miejsc, w których możemy zobaczyć zastosowanie tego pomysłu, jest kod rejestrujący metody jako programy obsługi zdarzeń w graficznym interfejsie użytkownika Tkinter (w Pythonie 2.x noszącym nazwę Tkinter). Poniżej znajduje się prosty przykład.

```

def handler():
    ...wykorzystanie zmiennych globalnych dla stanu...
    ...
widget = Button(text='mielonka', command=handler)

```

By zarejestrować program obsługi zdarzeń kliknięcia przycisku, zazwyczaj przekazujemy obiekt wywoływalny nieprzyjmujący argumentów do argumentu ze słowem kluczowym `command`. Działają tutaj nazwy funkcji (oraz `lambda`), a także metody klas, które muszą być wiązane, jeżeli w argumencie ma być umieszczona instancja:

```

class MyWidget:

    def handler(self):
        ...użycie self.attr dla stanu...

    def makewidgets(self):
        b = Button(text='mielonka', command=self.handler)

```

W powyższym kodzie programem obsługą zdarzeń jest `self.handler` — obiekt metody z wiązaniem pamiętający zarówno `self`, jak i `MyGui.handler`. Ponieważ kiedy metoda `handler` jest później wywoływana w momencie wystąpienia zdarzenia, metoda będzie miała dostęp do atrybutów instancji, które mogą przechowywać stan pomiędzy zdarzeniami, jak również do metod na poziomie klasy. W prostych funkcjach stan musiałby normalnie zostać zachowany w zmiennych globalnych lub w zakresie funkcji zewnętrznej.

Inny sposób tworzenia klas kompatybilnych z interfejsami API opartymi na funkcjach, polegający na przeciążaniu operatora `__call__`, jest opisany w rozdziale 30., natomiast w rozdziale 19. przedstawione jest inne narzędzie — funkcja `lambda` — często wykorzystywana do obsługi wywołań zwrotnych. Jak wcześniej wspomniałem, nie zawsze jest konieczne opakowywanie związanej metody w funkcję `lambda`. Metoda zdefiniowana w opisanym przykładzie odnacza już wywołanie (należy zwrócić uwagę na brak nawiasów), więc wprowadzenie jeszcze funkcji `lambda` jest bezcelowe.

Z technicznego punktu widzenia wszystkie klasy są również obiektami wywoływanymi, ale różnica polega na tym, że wywołuje się je w celu utworzenia instancji, a nie do wykonania określonej pracy. Wprawdzie lepiej jest prostą operację zakodować jako zwykłą funkcję niż klasę z konstruktorem, jednak w tym przykładzie klasa służy do zilustrowania cech obiektu wywoływanego.

```

>>> class Negate:

...     def __init__(self, val):                      # Klasa też są
    obiekta wywoływanymi

...         self.val = -val                          # Wywołanie z poziomu
obiektu nie zadziała

...     def __repr__(self):                         # Format reprezentacji
tekstowej

...         return str(self.val)

...

>>> actions = [square, sobject, pobject.method, Negate] # Wywołanie klasy
>>> for act in actions:
...     print(act(5))

...
25
7
15
-5

```

```

>>> [act(5) for act in actions]                                # Wywołuje
__repr__(), nie __str__()!
[25, 7, 15, -5]

>>> table = {act(5): act for act in actions}                # Składanie słownika
z 2.7/3.x

>>> for (key, value) in table.items():
...     print('{0:2} => {1}'.format(key, value))              # Formatowanie ciągu
znaków w 2.6+/3.x

...
25 => <function square at 0x...itd...>
15 => <bound method Product.method of <__main__.Product object at 0x...itd...>>
-5 => <class '__main__.Negate'>
7 => <__main__.Sum object at 0x...itd...>

```

Jak widzimy, metody związane, a ogólniej: model obiektów wykonywanych w Pythonie, to tylko wybrane dowody na to, że Python został zaprojektowany jako niezwykle elastyczny język programowania.

W tym momencie powinieneś już rozumieć model obiektów metod. Inne przykłady metod związanych można znaleźć w ramce „Znaczenie metod z wiązaniem oraz wywołań zwrotnych” oraz w poprzednim rozdziale — przy okazji dyskusji na temat metody `__call__`.

Klasy są obiektami — uniwersalne fabryki obiektów

Czasami w projektach opartych na klasach trzeba tworzyć obiekty w reakcji na pojawiające się wymagania, których nie da się przewidzieć w momencie pisania kodu. W takim przypadku przydaje się wzorzec projektowy znany pod nazwą fabryki. Dzięki dużej elastyczności Pythona fabryki mogą przyjmować różne formy, choć niektóre z nich wcale nie wyglądają na wyjątkowe.

Ponieważ klasy są obiektami, łatwo jest je przekazywać w programach czy przechowywać w strukturach danych. Można również przekazać klasy do funkcji generujących dowolne rodzaje obiektów. Takie funkcje nazywane są czasami w kręgach projektowania zorientowanego obiektowo *fabrykami* (ang. *factory*). Są one dużym wyzwaniem w językach z silnymi typami, takich jak C++, jednak w Pythonie są trywialne do zaimplementowania.

Funkcja składnia wywołania omówiona w rozdziale 18., pozwala w jednym kroku wywołać dowolną klasę z dowolną liczbą argumentów pozycyjnych lub kluczowych konstruktora w celu utworzenia dowolnej instancji^[2].

```

def factory(aClass, *pargs, **kargs):      # Krotka argumentów o zmiennej
    liczbie i słownik

        return aClass(*pargs, **kargs)          # Wywołanie konstruktora klasy
aClass (lub apply w 2.x)

class Spam:

    def doit(self, message):

```

```

    print(message)

class Person:

    def __init__(self, name, job=None):
        self.name = name
        self.job = job

    object1 = factory(Spam)           # Utworzenie obiektu Spam
    object2 = factory(Person, "Artur", "Król") # Utworzenie obiektu Person
    object3 = factory(Person, name='Bronek')   # Jak wyżej, z argumentem
                                                # kluczowym i domyślną wartością

```

W powyższym kodzie definiujemy funkcję generatora obiektów o nazwie `factory`. Oczekuje ona przekazania obiektu klasy (wystarczy dowolna klasa) wraz z jednym lub większą liczbą argumentów dla konstruktora klasy. Ta funkcja do wywołania konstruktora klasy używa specjalnej składni umożliwiającej wywoływanie funkcji i zwracanie instancji z dowolną listą parametrów.

Pozostała część przykładu definiuje po prostu dwie klasy i generuje ich instancje, przekazując je do funkcji `factory`. Jest to jedyna funkcja fabryki, jaką kiedykolwiek będziemy musieli pisać w Pythonie — działa ona dla każdej klasy i dowolnych argumentów konstruktora. Po uruchomieniu kodu (plik `factory.py`) obiekty będą wyglądały jak niżej:

```

>>> object1.doit(99)
99
>>> object2.name, object2.job
('Artur', 'Król')
>>> object3.name, object3.job
('Bronek', None)

```

Powinniśmy już wiedzieć, że w Pythonie wszystko jest obiektem, w tym klasy, które w językach takich jak C++ są tylko danymi wejściowymi kompilatora. Naturalną rzeczą jest więc przetwarzanie ich w taki sam sposób jak obiekty. Jak jednak wspomniano na początku tej części książki, jedynie obiekty *pochodzące* z klas są w Pythonie obiektami w pełnym rozumieniu programowania zorientowanego obiektowo.

Do czego służą fabryki

Do czego zatem mogą nam się przydać fabryki (poza dostarczeniem wymówki do zilustrowania obiektów klas w niniejszej książce)? Niestety, trudno jest pokazać zastosowanie tego wzorca projektowego bez zamieszczenia większej ilości kodu, niż mamy na to miejsce. Ogólnie rzecz biorąc, fabryki pozwalają na odizolowanie kodu od szczegółów konfigurowanej dynamicznie konstrukcji obiektu.

Przypomnijmy sobie przykład z klasą `processor`, zaprezentowany z abstrakcyjnego punktu widzenia w rozdziale 26. oraz w tym rozdziale jako przykład kompozycji. Przyjmował on obiekty typu `reader` oraz `writer` do przetwarzania dowolnych strumieni danych. Oryginalna wersja tego przykładu ręcznie przekazywała instancje wyspecjalizowanych klas, takich jak `FileWriter` oraz `SocketReader`, w celu dostosowania przetwarzanych strumieni danych do własnych potrzeb. Później przekazywaliśmy zapisane na stałe w kodzie obiekty pliku, strumienia oraz formatera. W bardziej dynamicznym scenariuszu do konfigurowania strumienia mogłyby

posłużyć narzędzia zewnętrzne, takie jak pliki konfiguracyjne czy graficzne interfejsy użytkownika.

W takim dynamicznym świecie być może nie uda nam się zapisać obiektów interfejsów strumieni na stałe w kodzie skryptu, jednak zamiast tego możemy je utworzyć w czasie wykonywania zgodnie z zawartością pliku konfiguracyjnego.

Plik może na przykład podawać po prostu łańcuch znaków nazwy klasy strumienia, jaka ma być zainportowana z modułu, wraz z opcjonalnym argumentem wywołania konstruktora. Mogą się tutaj przydać mechanizmy fabrykujące, ponieważ pozwolą nam pobrać oraz przekazać klasy, które nie są zapisane na stałe w naszym programie. Klasy te mogą nawet wcale nie istnieć w momencie pisania naszego kodu.

```
classname = ... odczytane z pliku konfiguracyjnego...
classarg = ... odczytane z pliku konfiguracyjnego...
import streamtypes                                     # Kod można dostosować do
własnych potrzeb
aclass = getattr(streamtypes, classname)             # Pobrane z modułu
reader = factory(aclass, classarg)                   # Lub aclass(classarg)
processor(reader, ...)
```

W powyższym kodzie wbudowana funkcja `getattr` jest ponownie wykorzystana do pobrania atrybutu modułu po podaniu łańcucha znaków nazwy (przypomina to kod `obiekt.atrybut`, ale `atrybut` jest tutaj łańcuchem znaków). Ponieważ ten fragment kodu zakłada istnienie jednego argumentu konstruktora, nie potrzebuje właściwie `factory` — moglibyśmy utworzyć instancję za pomocą kodu `aclass(classarg)`. Może się to jednak przydać w obecności nieznanej listy argumentów, a ogólny wzorzec tworzenia kodu fabryki może poprawić elastyczność kodu.

Dziedziczenie wielokrotne — klasy mieszane

Nasz ostatni wzorzec projektowy jest jednym z najbardziej przydatnych, dlatego na zakończenie tego rozdziału i jako zapowiedź następnego posłuży za podstawę dla bardziej praktycznego przykładu. Dodatkowo kod, który tu napiszemy, może okazać się przydatnym narzędziem.

Wiele projektów wykorzystujących klasy wykorzystuje osobne zestawy metod. Jak widzieliśmy, w instrukcji `class` w nawiasach znajdujących się w wierszu nagłówka można wymienić więcej niż jedną klasę nadrzędną. Jeśli tak zrobimy, wykorzystujemy coś, co nosi nazwę *dziedziczenia wielokrotnego* (ang. *multiple inheritance*) — klasa oraz jej instancje dziedziczą zmienne po *wszystkich* wymienionych klasach nadrzędnych.

Kiedy próbujemy odnaleźć atrybut, Python przeszukuje klasy nadrzędne z wiersza nagłówka klasy od lewej do prawej strony tak długo, aż nie znajdzie dopasowania. Ścisłe rzecz biorąc, proces wyszukiwania postępuje od dołu do góry drzewa dziedziczenia, a następnie od lewej strony do prawej, ponieważ każda klasa nadrzędną może mieć swoje własne klasy nadrzędne.

- W klasach „klasycznych” (stosowanych domyślnie przed wersją 3.0) wyszukiwanie nazw odbywało się metodą „w góre drzewa dziedziczenia, a następnie od lewej do prawej”. Kolejność ta nosi nazwę DFLR, od angielskiego *depth-first, left-to-right* (najpierw w głąb, potem od lewej na prawą).
- W klasach *nowego stylu* (co jest opcjonalne w Pythonie 2.x, a standardowe w 3.0) wyszukiwanie nazw odbywa się zazwyczaj tak samo jak wcześniej, choć w drzewach

diamondowych najpierw sprawdzane są poziomy wszerz, a potem następuje przejście w góre. Kolejność ta nosi nazwę *MRO w nowym stylu* (ang. *method resolution order*, kolejność odwzorowywania metod) i dotyczy nie tylko metod, ale też atrybutów.

Druga z powyższych reguł wyszukiwania zostanie szczegółowo opisana w następnym rozdziale poświęconym klasom w nowym stylu. Trudno jest ją opisać bez posilkowania się kodem z następnego rozdziału, ale w skrócie można powiedzieć, że *drzewo diamentowe* pojawia się wtedy, gdy kilka klas ma taką samą klasę nadzczną. Proces przeszukiwania klas w nowym stylu został zaprojektowany tak, aby wspólna klasa nadzczna została odwiedzona tylko raz, po odwiedzeniu wszystkich klas podrzędnych. Jednak w obu modelach, gdy dana klasa ma kilka klas nadzecznych, drzewo jest przeszukiwane od strony lewej na prawą zgodnie z kolejnością nazw w instrukcji `class`.

Dziedziczenie wielokrotne przydaje się do modelowania obiektów należących do więcej niż jednego zbioru. Osoba może na przykład być inżynierem, pisarzem oraz muzykiem i może dziedziczyć właściwości po wszystkich tych zbiorach. W przypadku wielokrotnego dziedziczenia obiekty uzyskują dostęp do wszystkich nazw zadeklarowanych w klasach nadzecznych. Jak się później przekonamy, dzięki wielokrotnemu dziedziczeniu klasy mogą pełnić role pakietów zawierających mieszane atrybuty.

Jest to przydatny wzorzec, jednak wielokrotne dziedziczenie ma ten poważny mankament, że jest podatne na *konflikty*, gdy w kilku klasach nadzecznych zostaną zdefiniowane metody (lub atrybuty) o takich samych nazwach. W takiej sytuacji konflikt jest rozwiązywany automatycznie poprzez przyjęcie kolejności przeszukiwania drzewa dziedziczenia lub ręcznie w następujący sposób:

- *Domyślnie*: podczas dziedziczenia wybierane jest *pierwsze wystąpienie atrybutu*, jeżeli odwołujemy się do niego w zwykły sposób, na przykład za pomocą instrukcji `self.metoda`. W takim przypadku w modelu klasycznym oraz niediamondowym wybierany jest atrybut zlokalizowany najbliżej, pierwszy z lewej. W przypadku klas w nowym stylu w modelu diamondowym przeszukiwanie odbywa się najpierw ze strony lewej na prawą, a potem w górę drzewa.
- *Jawnie*: w niektórych modelach klas trzeba jawnie *wskazywać atrybut* poprzez odwołanie się do niego za pomocą nazwy klasy, np. `klasa_nadzczna.metoda(self)`. Konflikt jest rozwiązywany w samym kodzie i ma pierwszeństwo przed domyślnym procesem przeszukiwania drzewa, tj. najpierw ze strony lewej na prawą, a potem w górę.

Opisany problem występuje tylko wtedy, gdy *ta sama nazwa* pojawia się w kilku klasach nadzecznych, a użycie pierwszej dziedziczonej nazwy nie jest pożądane. Ponieważ nie jest to często spotykany przypadek w typowym kodzie, odkładmy ten temat do następnego rozdziału, poświęconego klasom w nowym stylu, algorytmowi MRO i funkcji `super`. Będzie to również przykład pułapki opisany pod koniec następnego rozdziału. Teraz zajmijmy się praktycznym zastosowaniem narzędzi wielokrotnego dziedziczenia opisanym w następnym podrozdziale.

Tworzenie klas mieszanych

Chyba najpopularniejszym sposobem wykorzystywania dziedziczenia wielokrotnego jest mieszanie metod ogólnego przeznaczenia z klas nadzecznych. Takie klasy nadzędne nazywane są zazwyczaj *klasami mieszającymi* (ang. *mix-in class*) — udostępniają one metody dodawane do klas aplikacji za pomocą dziedziczenia. W pewnym sensie klasy mieszane są podobne do modułów, ponieważ stanowią pakiety metod przeznaczonych do wykorzystania w podrzędnych klasach klienckich. W przeciwnieństwie do prostych funkcji w modułach metody również tworzą hierarchię dziedziczenia i mają dostęp do instancji `self`, za pośrednictwem której mogą korzystać ze stanu i innych metod obiektu.

Jak widzieliśmy, domyślny dla Pythona sposób wyświetlania informacji o klasach nie jest szczególnie użyteczny:

```

>>> class Spam:
...     def __init__(self):                      # Nie ma __repr__
...         self.data1 = "jedzenie"
...
>>> X = Spam()
>>> print X                                # Wygląd domyślny: klasa, adres
<__main__.Spam instance at 0x00864818>>      # W Pythonie 2.x wyświetla
"instance"

```

Jak widzieliśmy w przykładzie w rozdziale 28. i w opisie przeciążania operatorów w rozdziale 30., możemy udostępnić własną metodę `__str__` lub `__repr__` implementującą naszą reprezentację tekstową. Zamiast jednak kodować jedną z powyższych metod w każdej klasie, którą chcemy wyświetlić, dlaczego nie zapisać jej raz w klasie narzędzia ogólnego przeznaczenia dziedziczonej przez wszystkie klasy?

Do tego właśnie służą klasy mieszane. Definiując na przykład metodę reprezentacji tekstowej w klasie nadzędnej, mamy możliwość wyświetlania informacji w niestandardowym formacie nawet w klasach pochodnych od innych klas nadzędnych. Poznaliśmy już narzędzia, które wykonują tego rodzaju operacje:

- rozdział 28.: klasa `AttrDisplay` formatowała atrybuty instancji klasy i wypisywała je za pomocą generycznej metody `__repr__`, ale nie zastosowaliśmy wówczas wielopoziomowej hierarchii klas, a jedynie pojedyncze dziedziczenie,
- rozdział 29.: moduł `classtree.py` definiował funkcje służące do nawigowania i rysowania schematów drzew dziedziczenia, ale kod ten nie wyświetlał atrybutów obiektów, nie był też zaprojektowany z myślą o dziedziczeniu.

Rozwińmy techniki prezentowane w tych przykładach, tworząc zestaw trzech klas mieszanych implementujących narzędzia do wyświetlania atrybutów obiektów, atrybutów dziedziczonych oraz atrybutów wszystkich obiektów w drzewie klas. Naszych narzędzi użyjemy w kontekście wielodziedziczenia, prezentując przy okazji techniki programowania dostosowane do tworzenia uogólnionych narzędzi.

Opisany przypadek można zakodować inaczej niż w rozdziale 28., tj. wykorzystując metodę `__str__`, a nie `__repr__`. Po części jest to błąd, który ogranicza rolę stylu do metod `print` i `str`, ale sposób wyświetlania danych będzie na tyle bogaty, że będzie można uznać go za przyjazny dla użytkownika. Ponadto będzie to przykład kodowania w klasach klienckich alternatywnego, niskopoziomowego wyświetlania interaktywnych odpowiedzi za pomocą zagnieżdżonej metody `__repr__`. Metoda ta umożliwia stosowanie alternatywnej metody `__str__`, a różnice pomiędzy obiema metodami zostały dokładniej opisane w rozdziale 30.

Odczyt listy atrybutów obiektu — `__dict__`

Zaczniemy od prostego przypadku: wypisywania atrybutów instancji. Poniższa klasa (zakodowana w pliku `listinstance.py`) definiuje klasę mieszzaną `ListInstance` przeciążającą metodę `__str__`. Ta metoda będzie wykorzystana we wszystkich klasach dziedziczących po `ListInstance`. Dzięki temu, że ta logika została zaimplementowana w postaci klasy, `ListInstance` jest uogólnionym narzędziem, którego funkcję można wykorzystać w dowolnej klasie potomnej.

```

#!/usr/bin/python
# Plik listinstance.py (wersje 2.x i 3.x)
class ListInstance:

```

```

"""
Klasa mieszana formatująca informacje wyświetlane przez metody
print i str za pomocą z kodowanej tutaj dziedziczonej metody __str__.
Nazwy __X zapobiegają konfliktom z atrybutami klas.
"""

def __attrnames(self):
    result = ''
    for attr in sorted(self.__dict__):
        result += '\tnazwa %s=%s\n' % (attr, self.__dict__[attr])
    return result

def __str__(self):
    return '<Instancja klasy %s, adres %s:\n%s>' % (
        self.__class__.__name__,      # nazwa klasy
        id(self),                   # adres
        self.__attrnames())          # lista
nazwa=wartość

if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListInstance)

```

Wszystkie kody prezentowane w tym podrozdziale działają poprawnie w wersjach 2.x i 3.x. Uwaga: w powyższym przykładzie użyte jest klasyczne wyrażenie. Ilość kodu można zmniejszyć, implementując metodę `__attrnames` i wyrażenie generatora wywoływanego za pomocą metody `join` użytej z ciągiem znaków. Jest to jednak mniej czytelne rozwiązanie. Zamiast stosować wyrażenie takie jak przedstawione niżej, warto rozważyć prostszy sposób kodowania:

```

def __attrnames(self):
    return ''.join('\t%s=%s\n' % (attr, self.__dict__[attr])
                  for attr in sorted(self.__dict__))

```

Klasa `ListInstance` wykorzystuje niektóre ze znanych nam już technik introspekcji nazwy klasy oraz atrybutów obiektu:

- Każda instancja obsługuje wbudowany atrybut `__class__` wskazujący klasę, z której została ona utworzona. Każda klasa obsługuje atrybut `__name__` określający jej nazwę, zatem odwołanie `self.__class__.__name__` zwraca nazwę klasy obiektu.
- Nasza klasa wykonuje większość swojej pracy, wykorzystując słownik atrybutów instancji (udostępniany w atrybucie `__dict__`), z którego buduje串 znaków wypisujący nazwy i wartości atrybutów. Klucze odczytane z tego słownika są wstępnie sortowane w celu uporządkowania wyników i uniezależnienia ich od wersji Pythona.

Pozostałe mechanizmy klasy `ListInstance` przypominają prezentowany w rozdziale 28. mechanizm wypisywania atrybutów — w zasadzie ten przykład można uznać za jedną z wariacji na temat tego zagadnienia. Warto jednak zwrócić uwagę na dwie cechy podejścia wykorzystanego w tej klasie:

- Wypisuje adres instancji w pamięci, wykorzystując wbudowaną funkcję `id` zwracającą adres dowolnego obiektu (jest to z definicji unikalny identyfikator obiektu, co przyda się w przyszłych modyfikacjach naszego kodu).
- Wykorzystuje wzorzec nazw *pseudoprivatnych* do obsługi metody `__attnames`. Jak dowiedzieliśmy się wcześniej w tym rozdziale, Python automatycznie modyfikuje tego typu nazwę w klasie nadzędnej, dopisując do niej nazwę klasy (w tym przypadku nazwa zmienia się na `_ListInstance__attnames`). Ta zasada obowiązuje w stosunku do atrybutów klas (również metod) oraz atrybutów instancji zapisanej w `self`. Jak wspomniałem w rozdziale 28., ten mechanizm przydaje się w przypadku ogólnych narzędzi, ponieważ pozwala się upewnić, że nazwy zdefiniowane w klasie nie kolidują z nazwami zdefiniowanymi w klasach potomnych.

Ponieważ klasa `ListInstance` definiuje metodę przeciążającą operator `__str__`, instancje tej klasy automatycznie wyświetlają nie tylko adresy atrybutów, ale też dodatkowe informacje. Poniższy listing prezentuje naszą klasę w działaniu, w sytuacji pojedynczego dziedziczenia (kod daje takie same wyniki w Pythonie 3.x i 2.x):

```
>>> from lister import ListInstance
>>> class Spam(ListInstance):           # Dziedziczy metodę __str__
...     def __init__(self):
...         self.data1 = 'jedzenie'
...
>>> x = Spam()
>>> print(x)                         # print i str wywołują __str__
<Instancja klasy Spam, adres 18931664:
    data1=jedzenie
>
```

Wynik powyższego kodu można przechwycić i zapisać jako串 znaków bez wyświetlania go za pomocą funkcji `str`, a rezultaty interaktywnych poleceń będą wykorzystywały domyślny format, ponieważ metoda `__repr__` jest opcjonalna dla klientów:

```
>>> display = str(x)      # Wyświetlenie w celu przechwycenia znaków ucieczki
>>> display
'<Instancja klasy Spam, adres 18931664:\n\tdatal=jedzenie\n>'
>>> x                      # Nadal domyślna jest metoda
__repr__
<__main__.Spam object at 0x000000000290A780>
```

Klasa `ListInstance` może być przydatna w dowolnych klasach tworzonych przez użytkownika, nawet w przypadku klas posiadających jedną lub kilka klas nadzędnych. W takich przypadkach *wielokrotne dziedziczenie* okazuje się szczególnie użyteczne: dodając `ListInstance` do listy dziedziczenia w deklaracji klasy (wmieszanie), otrzymujemy dostęp do zdefiniowanej w niej metody `__str__`, nie tracąc atrybutów dziedziczonych po pozostałych klasach. Właściwość tę demonstruje listing `testmixin.py`:

```
# Plik testmixin0.py
from listinstance import ListInstance      # Import klas modułu lister
class Super:
```

```

def __init__(self):                      # Metoda __init__ klasy nadzędnej
    self.data1 = 'mielonka'               # Tworzenie atrybutów instancji

def ham(self):
    pass

class Sub(Super, ListInstance):          # Wmieszanie metod ham i __str__
    def __init__(self):                  # Metody wyświetlające mają dostęp
        do self
            Super.__init__(self)
            self.data2 = 'jajka'           # Więcej atrybutów instancji
            self.data3 = 42
    def spam(self):                    # Definiujemy jeszcze jedną metodę
        pass

if __name__ == '__main__':
    X = Sub()
    print(X)                          # Wywołuje wmieszana metodę
__str__

```

Klasa Sub dziedziczy po klasach Super i ListInstance, w efekcie czego stanowi połączenie własnych nazw i nazw zdefiniowanych w klasach nadzędnych. Po utworzeniu instancji klasy Sub i wywołaniu na niej funkcji print automatycznie zostaje wyświetlona niestandardowa reprezentacja zdefiniowana w klasie ListInstance (w tym przypadku wynik skryptu jest identyczny w przypadku Pythona 3.x i 2.x, z wyjątkiem adresów obiektów, które oczywiście mogą być różne dla różnych procesów):

```

C:\code> python testmixin0.py
<Instancja klasy Sub, adres 44304144:
    data1=mielonka
    data2=jajka
    data3=42
>

```

Powyższy skrypt *testmixin0.py* działa poprawnie, ale nazwa testowanej klasy jest w nim zapisana na stałe, przez co eksperymentowanie z innymi klasami — czym zajmiemy się za chwilę — jest utrudnione. Aby program był bardziej elastyczny, można wykorzystać kod przeładowujący moduły, opisany w rozdziale 25., i przekazywać obiekt przeznaczony do sprawdzenia. Ilustruje to poniższy, ulepszony skrypt *testmixin.py*, wykorzystywany we wszystkich samotestujących kodach wyświetlających klasy. W tym kontekście obiekt przekazywany funkcji testującej jest *klasą* mieszaną, a nie funkcją, ale zasada pozostaje ta sama: wszystko w Pythonie jest obiektem, który można przekazywać.

```

#!/usr/bin/python
# Plik testmixin.py (wersje 2.x i 3.x)
"""

```

Generyczny tester wykorzystujący mieszane klasy. Podobny do

modułu przechodniego przeładowania z rozdziału 25., ale przekazujący do testera obiekt klasy, a nie funkcji. Funkcja `testByNames` umożliwia ładowanie za pomocą nazwy zarówno modułu, jak i klasy, zgodnie z wzorcem fabryki opisany w rozdziale 31.

```

"""
import importlib

def tester(listerclass, sept=False):
    class Super:
        def __init__(self):          # Metoda __init__ klasy nadzędnej
            self.data1 = 'mielonka'  # Utworzenie atrybutów instancji
        def ham(self):
            pass
    class Sub(Super, listerclass): # Zmieszanie nazw ham i __str__
        def __init__(self):          # Metoda listująca ma dostęp do self
            Super.__init__(self)
            self.data2 = 'jajka'      # Następne atrybuty instancji
            self.data3 = 42
        def spam(self):             # Zdefiniowanie innej metody
            pass
    instance = Sub()              # Zwrócenie instancji z metodą __str__
    klasy listującej
    print(instance)               # Uruchomienie zmieszczonej metody
    __str__ (lub poprzez str(x))
    if sept: print('-' * 80)

def testByNames(modname, classname, sept=False):
    modobject = importlib.import_module(modname)          # Import z użyciem
    nazwy
    listerclass = getattr(modobject, classname)           # Pobranie atrybutu
    z użyciem nazwy
    tester(listerclass, sept)

if __name__ == '__main__':
    testByNames('listinstance', 'ListInstance', True)   # Przetestowanie
    wszystkich trzech klas
    testByNames('listinherited', 'ListInherited', True)
    testByNames('listtree', 'ListTree', False)

```

Przy okazji powyższy skrypt umożliwia wskazanie modułu testowego i klasy za pomocą nazwy i wykorzystuje ten sposób w kodzie samotestującym. Jest to przykład zastosowania opisanego

wcześniej wzorca fabryki. Poniżej przedstawiony jest nowy skrypt w działaniu, uruchomiony za pomocą modułu listującego w celu przetestowania własnej klasy (wyniki są takie same w wersjach 2.x i 3.x). Skrypt można również uruchamiać niezależnie, jednak w tym trybie testowane są dwa warianty klasy listującej, których jeszcze nie widzieliśmy (i nie zakodowaliśmy!).

```
c:\code> python listinstance.py
<Instance of Sub, address 43256968:
    data1=mielonka
    data2=jajka
    data3=42
>
c:\code> python testmixin.py
<Instance of Sub, address 43977584:
    data1=mielonka
    data2=jajka
    data3=42
>
...Testy dwóch innych klas listujących za chwilę...
```

Klasa `ListInstance` działa z dowolną klasą potomną, ponieważ atrybut `self` odwołuje się do instancji tej klasy potomnej, z której wydobywają niezbędne informacje. W pewnym sensie klasy mieszane są odpowiednikiem modułów, czyli pakietów narzędzi, których mogą używać różni klienci. W poniższym przykładzie klasa `ListInstance` jest z powrotem użyta w trybie pojedynczego dziedziczenia z instancją innej klasy. Jest ładowana za pomocą instrukcji `import` i wyświetla atrybuty, którym wartości zostały przypisane poza klasą:

```
>>> import listinstance
>>> class C(listinstance.ListInstance): pass
...
>>> x = C()
>>> x.a, x.b, x.c = 1, 2, 3
>>> print(x)
<Instancja klasy C, adres 18922232:
    a=1
    b=2
    c=3
>
```

Oprócz wygody stosowania klasy mieszane pozwalają na optymalizację utrzymania kodu, jak wszystkie klasy. Jeśli jakiś czas po utworzeniu klasy zdecydujemy się na przykład zmodyfikować format ciągów znaków zwracanych z metodą `__str__` klasy `ListInstance` w taki sposób, aby zwracał również atrybuty klasy odziedziczone po klasach nadzędnych, nie będzie problemu. Dzięki temu, że metoda ta jest dziedziczona, wszystkie klasy dziedziczące po `ListInstance`

automatycznie skorzystają z tej zmiany. Uznajmy, że wspomniany „jakiś czas po utworzeniu klasy” właśnie minął i zajmijmy się tym zagadnieniem.

Wydobywanie atrybutów odziedziczonych z użyciem dir()

Nasza klasa `ListerInstance` wypisuje wyłącznie atrybuty instancji (to znaczy nazwy przypisane do samego obiektu instancji). Bardzo łatwo można jednak rozszerzyć ten mechanizm o wypisywanie wszystkich atrybutów dostępnych w instancji, zarówno własnych, jak i odziedziczonych po klasach nadrzędnych. W tym celu wystarczy skorzystać z funkcji wbudowanej `dir`, zamiast czytać ze słownika `__dict__`. Słownik `__dict__` zawiera bowiem jedynie atrybuty instancji, natomiast funkcja `dir` dodatkowo zwraca atrybuty odziedziczone (od Pythona 2.2 wzwyż).

Poniższy listing prezentuje ten schemat programowania. Aby uprościć testowanie, umieściłem kod w osobnym module, ale gdyby obecni klienci musieli użyć tej właśnie wersji klasy, wybórą ją automatycznie (jak pamiętamy z rozdziału 25., za pomocą klauzuli `as` w instrukcji `import` można zmieniać nazwę nowej klasy na wcześniej używaną):

```
#!/usr/bin/python

# Plik listinherited.py (wersje 2.x i 3.x)

class ListInherited:
    """
    Wykorzystujemy funkcję dir() do uzyskania listy atrybutów instancji
    oraz atrybutów odziedziczonych. W Pythonie 3.x uzyskamy większą liczbę
    nazw w porównaniu z 2.x, ponieważ w modelu klas w nowym stylu niejawnie
    stosowana jest klasa nadrzędna super. Metoda getattr pobiera
    odziedziczone

    nazwy, których nie ma w self.__dict__. Zamiast metody __repr__ należy
    użyć __str__, ponieważ w przeciwnym wypadku podczas wyświetlania
    związanych metod nastąpi zapętlenie!

    """

    def __attrnames(self):
        result = ''
        for attr in dir(self):                                # Funkcja dir
            if attr[:2] == '__' and attr[-2:] == '__':          # Pominiecie
                wewnętrznych nazw
                    result += '\t%s\n' % attr
            else:
                result += '\t%s=%s\n' % (attr, getattr(self, attr))
        return result

    def __str__(self):
        return '<Instance of %s, address %s:\n%s>' % (
```

```

        self.__class__.__name__,           # Własna nazwa
klasy
        id(self),                      # Własny adres
        self.__attrnames())            # nazwa=lista
wartości
if __name__ == '__main__':
    import testmixin
    testmixin.tester(ListInherited)

```

Warto zwrócić uwagę, że pomijamy nazwy typu `__X__`. Większość z takich nazw dotyczy atrybutów używanych w sposób niejawny (wewnętrznie), przez co raczej nie będą interesujące w zastosowaniach, do których napisaliśmy naszą klasę. W tej wersji zamiast indeksować słownik atrybutów instancji, musimy użyć wbudowanej funkcji `getattr` pobierającej atrybuty za pomocą nazwy i wykorzystującej protokół przeszukiwania dziedziczonych nazw, ponieważ niektóre nazwy nie są zapisane w samej instancji.

Aby sprawdzić w działaniu naszą nową klasę, należy uruchomić powyższy plik w zwykły sposób. Zdefiniowana w nim klasa jest przekazywana do funkcji testowej zapisanej w pliku `testmixin.py` i jest wykorzystywana jako podrzędna klasa mieszana. Wynik działania klas testowej i wyświetlającej będzie się różnił dla różnych wersji Pythona, ponieważ funkcja `dir` zwraca inne wyniki. W 2.x uzyskamy wynik przedstawiony na poniższym listingu. Należy zwrócić uwagę na efekt działania mechanizmu zniekształcania nazw (wynik został przycięty, aby zajmował mniej miejsca).

```

C:\code> c:\python27\python listinherited.py
<Instancja klasy Sub, adres 35161352:
    _ListInherited__attrnames=<bound method Sub.__attrnames of <test...itd...
>>
    __doc__
    __init__=<=
    __module__=<=
    __str__=<=
    data1=mielonka
    data2=jajka
    data3=42
    ham=<bound method Sub.ham of <testmixin.Sub instance at 0x0...itd...>>
    spam=<bound method Sub.spam of <testmixin.Sub instance at 0x...itd...>>
>

```

W Pythonie 3.x skrypt wypisze więcej nazw, ponieważ wszystkie klasy „nowego typu” dziedziczą nazwy po klasie `object` (więcej szczegółów na ten temat przedstawię w rozdziale 32.). Sporo tych nazw jest dziedziczonych po domyślnej klasie nadrzędnej, dlatego wiele z nich pominąłem (w wersji 3.3 są 32 nazwy). Aby uzyskać pełną listę, należy samodzielnie uruchomić ten kod.

```

C:\code> c:\python33\python listinherited.py
<Instancja klasy Sub, adres 43253152:

```

```

>>     __ListInherited__attrnames=<bound method Sub.__attrnames of test...itd...
      __class__
      __delattr__
      __dict__
      __doc__
      __eq__
      ...pominięte 32 nazwy...
      __repr__
      __setattr__
      __sizeof__
      __str__
      __subclasshook__
      __weakref__
      data1=mielonka
      data2=jajka
      data3=42
      ham=<bound method Sub.ham of <testmixin.tester.<locals>.Sub ...itd...>>
      spam=<bound method Sub.spam of <testmixin.tester.<locals>.Sub ...itd...>>
>

```

Jednym z potencjalnych rozwiązań problemu rozpowszechnienia odziedziczonych wbudowanych nazw i długich wartości jest przedstawiona niżej alternatywna wersja funkcji `__attrnames` (zawarta w pliku `listinherited2.py`), która umieszcza w osobnej grupie nazwy zawierające podwójne znaki podkreślenia, przez co minimalizuje zawijanie wierszy w przypadku dużych wartości atrybutów. Warto zwrócić uwagę na kodowanie znaku % w postaci sekwencji %%, dzięki której jest on zachowywany w końcowej operacji formatowania wyniku:

```

def __attrnames(self, indent=' '*4):
    result = 'Podkreślenia%s\n%s%%s\nInne%s\n' % ('-'*77, indent, '-'*77)
    unders = []
    for attr in dir(self):                                # Metoda dir instancji
        if attr[:2] == '__' and attr[-2:] == '__':        # Pominięcie
            wewnętrznych nazw
                unders.append(attr)
        else:
            display = str(getattr(self, attr))[:82-(len(indent) + len(attr))]
            result += '%s%s=%s\n' % (indent, attr, display)
    return result % ', '.join(unders)

```

Po tej zmianie wynik testu klasy jest nieco bardziej skomplikowany, ale też bardziej zwięzły i przydatny:

```
c:\code> c:\python27\python listinherited2.py
<Instancja klasy Sub, adres 36299208:
Podkreślenia-----
-----
    __doc__, __init__, __module__, __str__
Inne-----
-----
    _ListInherited__attrnames=<bound method Sub.__attrnames of <testmixin.Sub
insta
        data1=mielonka
        data2=jajka
        data3=42
        ham=<bound method Sub.ham of <testmixin.Sub instance at
0x000000000229E1C8>>
        spam=<bound method Sub.spam of <testmixin.Sub instance at
0x000000000229E1C8>>
    >
c:\code> c:\python33\python listinherited2.py
<Instancja klasy Sub, adres 43318912:
Podkreślenia-----
-----
    __class__, __delattr__, __dict__, __dir__, __doc__, __eq__, __format__,
__ge__,
    __getattribute__, __gt__, __hash__, __init__, __le__, __lt__, __module__,
__ne__,
    __new__, __qualname__, __reduce__, __reduce_ex__, __repr__, __setattr__,
__sizeof__,
    __str__, __subclasshook__, __weakref__
Inne-----
-----
    _ListInherited__attrnames=<bound method Sub.__attrnames of
<testmixin.tester.<l
        data1=mielonka
        data2=jajka
        data3=42
        ham=<bound method Sub.ham of <testmixin.tester.<locals>.Sub object at
0x00000000
```

```
spam=<bound method Sub.spam of <testmixin.tester.<locals>.Sub object at  
0x000000  
>
```

Formatowanie wyświetlanych danych to otwarty problem (np. można tu wykorzystać standardowy moduł `pprint` — ang. *pretty printer*, czyli zgrabny wydruk), więc dodatkowe cyzelowanie wyniku pozostawiam czytelnikom jako ćwiczenie. Kod wyświetlający strukturę drzewa na pewno i tak będzie bardziej przydatny.



Pętla w metodzie `_repr_`. Jedna przestroga: w związku z tym, że wypisujemy również odziedziczone metody, musimy przeciągać metodę `_repr_` zamiast `_str_`. W przeciwnym razie ten kod spowodowałby *zapętlanie*: wypisywanie wartości metody wywołuje metodę `_repr_` klasy metody. Innymi słowy, jeśli metoda `_repr_` zdefiniowana w klasie spróbuje wypisać metodę, spowoduje to ponowne wywołanie metody `_repr_` tej klasy. Różnica jest subtelna, lecz istotna. Jeśli ktoś chciałby się przekonać, wystarczy zmienić nazwę metody `_str_` na `_repr_`. Jeśli ktoś jest zmuszony do wykorzystania metody `_repr_` w takim kontekście, może uniknąć pętli, porównując typ wartości atrybutu za pomocą funkcji `isinstance` z wartością `types.MethodType` ze standardowej biblioteki. W ten sposób można sprawdzać, które elementy da się pominąć.

Wypisywanie atrybutów dla każdego obiektu w drzewie klas

Zaprogramujmy ostatnie już rozszerzenie klasy wypisującej atrybuty. W obecnej postaci klasa nie informuje o tym, z której klasy w hierarchii dziedziczenia pochodzi dany argument. Jak widzieliśmy w przykładzie `classtree.py` z rozdziału 29., w kodzie mamy możliwość przechodzenia w hierarchii klas. Poniższy listing (plik `listtree.py`) prezentuje klasę mieszaną wykorzystującą tę technikę do wyświetlania atrybutów pogrupowanych po klasach, w których zostały zdefiniowane. To w efekcie daje pełny szkic drzewa dziedziczenia, wyświetlając atrybuty każdej z klas. Wciąż trzeba domniemywać dziedziczenia atrybutów, ale uzyskuje się znacznie więcej szczegółów niż tylko zwykłą płaską listę:

```
#!python  
  
# Plik listtree.py (wersje 2.x i 3.x)  
  
class ListTree:  
  
    """  
  
        Klasa mieszana zwracająca ślad _str_ całego drzewa klasy  
        i atrybutów wszystkich jego obiektów na poziomie self i powyżej.  
        Metoda str uruchamiana za pomocą funkcji print zwraca skonstruowany  
        ciąg. Klasa wykorzystuje nazwy _X atrybutów, aby uniknąć konfliktów  
        z klientami. Jawnie odwołuje się do klas nadrzędnych i dla przejrzystości  
        wykorzystuje metodę str.format.  
    """  
  
    def __attrnames(self, obj, indent):  
        spaces = ' ' * (indent + 1)  
        result = ''
```

```

for attr in sorted(obj.__dict__):
    if attr.startswith('__') and attr.endswith('__'):
        result += spaces + '{0}\n'.format(attr)
    else:
        result += spaces + '{0}={1}\n'.format(attr, getattr(obj,
attr))
return result

def __listclass(self, aClass, indent):
    dots = '.' * indent
    if aClass in self.__visited:
        return '\n{0}<Klasa {1}: adres {2}: (Patrz wyżej)>\n'.format(
            dots,
            aClass.__name__,
            id(aClass))
    else:
        self.__visited[aClass] = True
        here = self.__attrnames(aClass, indent)
        above = ''
        for super in aClass.__bases__:
            above += self.__listclass(super, indent+4)
        return '\n{0}<Klasa {1}, adres {2}:\n{3}{4}{5}>\n'.format(
            dots,
            aClass.__name__,
            id(aClass),
            here, above,
            dots)

def __str__(self):
    self.__visited = {}
    here = self.__attrnames(self, 0)
    above = self.__listclass(self.__class__, 4)
    return '<Instancja {0}, adres {1}:\n{2}{3}>'.format(
        self.__class__.__name__,
        id(self),
        here, above)

if __name__ == '__main__':

```

```

import testmixin
testmixin.tester(ListTree)

```

Powyższa klasa osiąga zamierzony cel, przemierzając drzewo dziedziczenia od atrybutu `__class__` instancji do jej klasy, a następnie rekurencyjnie od atrybutu `bases` klasy do wszystkich klas nadrzędnych, skanując przy tym atrybut `dict` każdego obiektu. Ostatecznie, po zakończeniu rekurencji, łączy ze sobą wszystkie fragmenty ciągów znaków.

Zrozumienie rekurencyjnego programu, takiego jak powyższy, zajmuje trochę czasu, jednak ze względu na fakt, że drzewo dziedziczenia może mieć dowolny kształt i wysokość, nie można było tego rozwiązać w inny sposób (można ewentualnie zastosować jawne sortowanie, które poznaliśmy w rozdziałach 19. i 25., jednak nie byłby to prostszy sposób, dlatego ze względu na ograniczony czas i miejsce jego opis został pominięty). Powyższa klasa została napisana w najbardziej jawnym i przejrzystym sposobie.

Można na przykład w metodzie `_listclass` zastąpić instrukcję pętli przedstawioną w pierwszym fragmencie poniżej wyrażeniem niejawnie uruchamiającym generator, pokazanym w drugim fragmencie. Jednak w tym przypadku *rekurencyjne wywołania osadzone w wyrażeniu generatora* stanowią zbyt zawiłe rozwiązanie, a jego wydajność jest wątpliwa, zważywszy na ograniczony zakres programu. Żadne z alternatywnych rozwiązań nie tworzy tymczasowej listy, chociaż pierwsze może tworzyć tymczasowe wyniki w zależności od wewnętrznej implementacji ciągów, konkatenacji i metody `join`. Należało tutaj użyć narzędzi opisanych w rozdziale 21. do mierzenia czasu wykonania kodu.

```

above = ''
for super in aClass.__bases__:
    above += self._listclass(super, indent+4)
...lub...
above = ''.join(
    self._listclass(super, indent+4) for super in aClass.__bases__)

```

Można również w metodzie `_listclass` zakodować klauzulę `else`, jak we wcześniejszych wydaniach książki (patrz niżej). Jest to alternatywne rozwiązanie osadzające wszystko w liście argumentów metody `format`. Wykorzystywany jest tu fakt, że metoda `join` uruchamia wyrażenie operatora i jego rekurencyjne wywołania, zanim jeszcze w operacji formatowania rozpoczęnie się tworzenie wynikowego tekstu. Jest to bardziej skomplikowany sposób, choć sam go napisałem (to nie jest dobry znak!).

```

self._visited[aClass] = True
genabove = (self._listclass(c, indent+4) for c in aClass.__bases__)
return '\n{0}<Klasa {1}, adres {2}:\n{3}{4}>\n'.format(
    dots,
    aClass.__name__,
    id(aClass),
    self._attrnames(aClass, indent), # Uruchomienie
przed metodą format!
    ''.join(genabove),
    dots)

```

Jak zawsze, jawność jest lepsza niż niejawność, a tworzony kod może mieć tu równie duże znaczenie, jak wykorzystywane w nim narzędzia.

Warto zwrócić uwagę, że w tej wersji wykorzystywana jest właściwa dla wersji Pythona 3.x i 2.6/2.7 metoda `format`, a nie wyrażenie z operatorem `%`. Dzięki temu operacje podstawiania są bardziej zrozumiałe, ponieważ przy dużej liczbie podstawień, jak w naszym przykładzie, kod staje się znacznie czytelniejszy. Innymi słowy, ten sam efekt można uzyskać, wywołując jedno z poniższych wyrażeń — my wybraliśmy tę drugą formę:

```
return '<Instancja klasy %s, adres %s:\n%s%s>' % (...)          # Wyrażenie
return '<Instancja klasy {0}, adres {1}:\n{2}{3}>'.format(...)      # Metoda
```

Ten sposób ma niestety pewne mankamenty ujawniające się w wersjach 3.2 i 3.3. Aby je poznać, trzeba uruchomić kod.

Uruchomienie kodu wyświetlającego drzewo

Aby przetestować kod, należy uruchomić plik w taki sam sposób jak wcześniej. Klasa `ListTree` zostanie przekazana do pliku `testmixin.py` w celu zmieszania jej z klasą podrzędną w funkcji testowej. Wynik kodu wyświetlającego szkic drzewa w Pythonie 2.x wygląda następująco:

```
C:\code> c:\python27\python listtree.py
<Instancja klasy Sub, adres 36690632:
    _ListTree_visited={}
        data1=mielonka
        data2=jajka
        data3=42
....<Klasa Sub, adres 36652616:
    __doc__
    __init__
    __module__
    spam=<unbound method Sub.spam>
.....<Klasa Super, adres 36652712:
    __doc__
    __init__
    __module__
    ham=<unbound method Super.ham>
.....>
.....<Klasa ListTree, adres 30795816:
    _ListTree_attrnames=<unbound method ListTree.__attrnames>
    _ListTree_listclass=<unbound method ListTree.__listclass>
    __doc__
    __module__
    __str__
```

```
.....>
....>
>
```

Po uruchomieniu kodu w wersji 2.x w wynikach widać *niezwiązane* metody, ponieważ odczytujemy je bezpośrednio z klas. Wersja przedstawiona w poprzednim podrozdziale wyświetlała je jako metody *związane*, ponieważ klasa `ListInherited` pobierała je z *instancji* za pomocą metody `getattr` (pierwsza wersja indeksowała atrybut `_dict_` instancji i nie wyświetlała w ogóle odziedziczonych metod). Widzimy również efekt zniekształcania nazw na atrybutie `_visited`. Jeżeli tylko nie będziemy mieć wyjątkowego pecha, unikniemy w ten sposób konfliktu z innymi danymi. Niektóre metody klasy również zostały zniekształcone w celu uzyskania pseudoprzywatności.

Po uruchomieniu tego modułu w Pythonie 3.x ponownie uzyskamy znacznie większą liczbę atrybutów, która może się różnić w zależności od wersji. Jak się dowiemy w następnym rozdziale, w wersji 3.x wszystkie klasy na najwyższym poziomie automatycznie dziedziczą cechy wbudowanej klasy `object`. W Pythonie 2.x należy takie klasy kodować ręcznie, jeżeli pożądane są funkcjonalności właściwe klasom w nowym stylu. Należy również zwrócić uwagę, że niezwiązane metody w wersji 2.x są w wersji 3.x zwykłymi *funkcjami*. Część wyniku pominąłem w celu zaoszczędzenia miejsca; warto uruchomić ten kod samodzielnie, aby przeanalizować efekt działania w całości.

```
C:\code> c:\python33\python listtree.py
<Instancja klasy Sub, adres 44277488:
    _ListTree__visited={}
        data1=mielonka
        data2=jajka
        data3=42
....<Klasa Sub, adres 36990264:
    __doc__
    __init__
    __module__
    __qualname__
    spam=<function tester.<locals>.Sub.spam at 0x000000002A3C840>
.....<Klasa Super, adres 36989352:
    __dict__
    __doc__
    __init__
    __module__
    __qualname__
    __weakref__
    ham=<function tester.<locals>.Super.ham at 0x000000002A3C730>
.....<Klasa object, adres 505114624:
```

```

__class__
__delattr__
__doc__
__eq__
...pominięte 2.x nazwy...
__repr__
__setattr__
__sizeof__
__str__
__subclasshook__

....>
....>
.....<Klasa ListTree, adres 36988440:
    _ListTree__attrnames=<function ListTree.__attrnames at
0x0000000002A3C158>
    _ListTree__listclass=<function ListTree.__listclass at
0x0000000002A3C1E0>
        __dict__
        __doc__
        __module__
        __qualname__
        __str__
        __weakref__
.....<Klasa object:, adres 505114624: (patrz wyżej)>
....>
....>
>

```

W tej wersji odwiedzona klasa nie jest wyświetlana dwa razy, ponieważ tworzona jest tabela odwiedzonych klas (dlatego właśnie dołączony jest identyfikator obiektu, który służy jako klucz elementu wyświetlonego już w raporcie). Podobnie jak w opisany w rozdziale 25. kodzie do przejściowego przeładowywania modułu, do zapobiegania powtórzeniom w wyświetlanych wynikach wykorzystywany jest słownik. Obiekty klas są kodowane i mogą służyć jako klucze w słowniku. Podobną funkcjonalność oferuje zbiór.

Z technicznego punktu widzenia w drzewie dziedziczenia *zapętleń* nie są możliwe. Klasa musi być zdefiniowana, aby mogła być nazwaną klasą nadzczną. Przy próbie utworzenia zapętleń poprzez zmianę atrybutu `__bases__` zostanie zgłoszony wyjątek. Jednak zastosowany tu mechanizm zapobiega dwukrotnemu wyświetleniu klasy.

```
>>> class C: pass
```

```
>>> class B(C): pass
>>> C.__bases__ = (B,)           # Głęboka, czarna magia!
TypeError: a __bases__ item causes an inheritance cycle
```

Inny przykład użycia: wyświetlenie nazw zawierających znaki podkreślenia

W tym przypadku również staraliśmy się unikać wyświetlania dużej liczby obiektów wewnętrznych, pomijając wartości dla nazw typu `__X__`. Gdyby zamienić w komentarz kod, który w specjalny sposób traktuje tego rodzaju nazwy atrybutów, wtedy ich wartości zostałyby wyświetcone w zwykły sposób:

```
for attr in sorted(obj.__dict__):
    #     if attr.startswith('__') and attr.endswith('__'):
    #         result += spaces + '{0}\n'.format(attr)
    #     else:
    #         result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
```

Poniżej przedstawiony jest wynik uzyskany w wersji 2.x po tymczasowym wprowadzeniu powyższej zmiany. Widoczne są wartości wszystkich atrybutów w drzewie klas:

```
c:\code> c:\python27\python listtree.py
<Instancja klasy Sub, adres 35750408:
 _ListTree_visited={}
 data1=mielonka
 data2=jajka
 data3=42
 ....<Klasa Sub, adres 36353608:
      __doc__=None
      __init__=<unbound method Sub.__init__>
      __module__=testmixin
      spam=<unbound method Sub.spam>
 .......<Klasa Super, adres 36353704:
      __doc__=None
      __init__=<unbound method Super.__init__>
      __module__=testmixin
      ham=<unbound method Super.ham>
 .......>
 .....<Klasa ListTree, adres 31254568:
      _ListTree_attrnames=<unbound method ListTree.__attrnames>
      _ListTree_listclass=<unbound method ListTree.__listclass>
```

```

__doc__=
Mieszana klasa zwracająca ślad __str__ całego drzewa klas i wszystkich
atrybutów ich obiektów na poziomie self i powyżej. Uruchamiana
za pomocą funkcji print lub str zwraca ciąg znaków. Wykorzystuje
nazwy atrybutów __X, aby nie kolidować z klientami. Jawnie
i rekurencyjnie przetwarza klasy nadrzędne. Wykorzystuje funkcję
str.format do czytelnego wyświetlania wyniku.

__module__=__main__
__str__=<unbound method ListTree.__str__>
.....
....>
....>
>

```

W wersji 3.x wynik powyższego testu jest znacznie obszerniejszy, co uzasadnia zaimplementowane wcześniej odrzucanie nazw zawierających znak podkreślenia. W rzeczywistości ten test może nawet nie wykonać się w niektórych najnowszych wersjach Pythona, np.:

```

c:\code> c:\python33\python listtree.py
...itd...
File "listtree.py", line 18, in __attrnames
result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
TypeError: Type method_descriptor doesn't define __format__

```



Diagnozowanie problemu z metodą str.format. Kod zamieniony w komentarz działa poprawnie w wersjach 3.0 i 3.1, więc problem pojawiający się w wersjach 3.2 i 3.3 wydaje się błędem lub przynajmniej regresem. Pojawia się wyjątek, ponieważ pięć wbudowanych metod nie definiuje atrybutu __format__ wymaganego przez funkcję str.format, a domyślny atrybut w obiekcie prawdopodobnie nie jest już poprawnie stosowany, gdy celem jest puste lub generyczne formatowanie. Aby się o tym przekonać, wystarczy uruchomić uproszczony kod, który izoluje ten problem:

```

c:\code> py -3.1
>>> '{0}'.format(object.__reduce__)
<method '__reduce__' of 'object' objects>
c:\code> py -3.3
>>> '{0}'.format(object.__reduce__)
TypeError: Type method_descriptor doesn't define __format__

```

Zgodnie z uzyskanymi wcześniej wynikami i aktualną dokumentacją Pythona puste cele, jak użyte w tym przypadku, powinny przekonwertować obiekt na jego wyświetlany ciąg (patrz oryginalna dokumentacja PEP 3101 i podręcznik do wersji 3.3). Co ciekawe, cele {0} i {0:s} powodują błąd, natomiast {0!s} wymusza poprawną konwersję, podobnie jak ręczna wstępna konwersja.

Odwziewciedla to zmianę dla przypadku specyficznego dla typu, w którym pomijane są częściej stosowane tryby użytkowania:

```
c:\code> py -3.3
>>> '{0:s}'.format(object.__reduce__)
TypeError: Type method_descriptor doesn't define __format__
>>> '{0!s}'.format(object.__reduce__)
"<method '__reduce__' of 'object' objects>"
>>> '{0}'.format(str(object.__reduce__))
"<method '__reduce__' of 'object' objects>"
```

Aby rozwiązać problem, należy wywołanie formatujące opakować w instrukcję `try` i przechwytywać wyjątek. Zamiast metody `str.format` można użyć wyrażenia formatującego `%`. Można również użyć jednego ze wspomnianych wcześniej, cały czas poprawnie działających trybów użycia metody `str.format` z nadzieją, że się nie zmienią. W ostateczności można zaczekać na poprawienie błędu w kolejnej wersji 3.x. Poniżej przedstawione jest zalecane rozwiązanie, w którym wykorzystane jest sprawdzone i niezawodne wyrażenie `%` (które na dodatek jest krótsze, jednak nie będę tutaj powtarzał porównań z rozdziału 7.).

```
c:\code> py -3.3
>>> "%s" % object.__reduce__
"<method '__reduce__' of 'object' objects>"
```

Aby zastosować powyższe rozwiązanie w kodzie wyświetlającym klasy, należy zmienić pierwszy poniższy wiersz na drugi:

```
result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
result += spaces + '%s=%s\n' % (attr, getattr(obj, attr))
```

Ten sam regres pojawia się w wersji 2.7 (prawdopodobnie odziedziczony po 3.2), ale nie w 2.6. Nie są wyświetlane metody obiektów w przykładzie z tego rozdziału. Ponieważ przykładowy kod uruchomiony w wersji 3.x generuje zbyt obszerny wynik, można go kwestionować, ale jest to dobry przykład rzeczywistego kodowania. Niestety, nowe funkcjonalności, takie jak metoda `str.format`, narzucają czasami programiście niewidoczną rolę testera aktualnej wersji 3.x!

Analizowałem zmianę kodu Pythona, aby rozwiązać ten problem, ale można go potraktować jako przykład diagnozowania wymagań i technik stosowanych w dynamicznym i otwartym projekcie, jakim jest Python. Zgodnie z poniższą uwagą wywołanie `str.format` nie obsługuje już określonego typu obiektów będących wartościami wbudowanych atrybutów. Jest to kolejny powód, aby pomijać ich nazwy.

Inny przykład użycia: uruchomienie kodu z większymi modułami

Ciekawe efekty daje usunięcie znaków komentarza z wierszy przetwarzających znaki podkreślenia i mieszanie metod naszej klasy z bardziej rozbudowaną klasą, jak `Button` z modułu `tkinter`. W nagłówku instrukcji `class` trzeba jako pierwszą umieścić nazwę `ListTree`, aby została wybrana jej metoda `__str__`. Klasa `Button` też ma tę metodę, ale klasa nadziedziona umieszczona jako pierwsza z lewej jest w dziedziczeniu wielokrotnym przeszukiwana jako pierwsza.

Wynik uruchomienia poniższego kodu jest ogromny (w wersji 3.x zajmuje 20 tys. znaków i 330 wierszy, a jeżeli zapomni się o usunięciu komentarzy, będzie to 38 tys. znaków). Dlatego chcąc zobaczyć pełny wynik, należy samodzielnie uruchomić ten kod. Warto zwrócić uwagę, jak słownik `_visited` mieszka się niegroźnie z atrybutami utworzonymi przez sam moduł `tkinter`. Jeżeli wykorzystywana jest wersja 2.x, należy użyć nazwy modułu `Tkinter`, a nie `tkinter`.

```
>>> from lister import ListTree
>>> from tkinter import Button
>>> class MyButton(ListTree, Button):
...     __str__ = ...
...     def __str__(self):
...         return self._name
...
>>> B = MyButton(text='mielonka')
>>> open('savetree.txt', 'w').write(str(B))
# Obie klasy definiują metodę __str__
# Najpierw ListTree: używamy __str__
...
20513
>>> len(open('savetree.txt').readlines())
# Zapis wyniku do pliku do późniejszego przejrzenia
330
>>> print(B)
# Wiersze w pliku
<Instancja klasy MyButton, adres 43363688:
    _ListTree__visited={}
    _name= 43363688
    _tclCommands=[]
    _w=.43363688
    children={}
    master=.
    ...
    ...pominięty duży fragment wyniku...
>
>>> S = str(B)
# Wyświetlenie tylko pierwszej części
>>> print(S[:1000])
```

Warto samodzielnie poeksperymentować w opisany sposób. Celem tego ćwiczenia było wykazanie, że w programowaniu zorientowanym obiektowo chodzi przede wszystkim o ponowne użycie kodu, a klasy mieszane są doskonałą ilustracją tej właściwości. Jak większość koncepcji programowania obiektowego, wielokrotne dziedziczenie może być użytecznym narzędziem, jeśli jest użyte prawidłowo. W praktyce jednak należy pamiętać, że to zaawansowana cecha języka i nieuwazne lub nieuzasadnione jej stosowanie może szybko doprowadzić do niepotrzebnej komplikacji. Wróćmy do tego problemu na końcu następnego rozdziału.

Moduł kolektora

Na koniec, aby importowanie naszych narzędzi było jeszcze prostsze, można utworzyć moduł kolektora, który umieści wszystkie narzędzia w jednej przestrzeni nazw. Po zaimportowaniu

poniższego modułu od razu uzyska się dostęp do wszystkich trzech mieszanych klas wyświetlających informacje:

```
# Plik lister.py

# Umieszczenie dla wygody wszystkich trzech klas w jednym module

from listinstance import ListInstance

from listinherited import ListInherited

from listtree import ListTree

Lister = ListTree                                # Wybór domyślnej klasy
```

W kodzie importującym poszczególne klasy mogą być wykorzystywane w takiej postaci, w jakiej są. Ewentualnie w instrukcji `import` można im nadawać zamienne nazwy do wykorzystania w klasach podrzędnych:

```
>>> import lister

>>> lister.ListInstance                         # Wybranie określonej klasy

<class 'listinstance.ListInstance'>

>>> lister.Lister                               # Użycie domyślnej klasy

Lister

<class 'listtree.ListTree'>

>>> from lister import Lister                  # Użycie domyślnej klasy

Lister

>>> Lister

<class 'listtree.ListTree'>

>>> from lister import ListInstance as Lister   # Użycie aliasu klasy Lister

>>> Lister

<class 'listinstance.ListInstance'>
```

W Pythonie tworzenie elastycznych interfejsów API dla różnych narzędzi odbywa się niemal automatycznie.

Miejsce na udoskonalenia: algorytm MRO, sloty, interfejsy graficzne

W tym przykładzie, tak jak w każdym programie, można osiągnąć znacznie więcej. Poniżej przedstawionych jest kilka propozycji rozszerzeń do ewentualnego przeanalizowania. Niektóre z nich są ciekawymi projektami. Dwie propozycje stanowią wprowadzenie do następnego rozdziału, jednak ze względu na ograniczone miejsce muszą pozostać jako proponowane ćwiczenia.

Ogólne pomysły: interfejsy graficzne, narzędzia wbudowane

Grupując nazwy zawierające podwójne znaki podkreślenia, jak to zrobiliśmy, można zmniejszyć ilość wyświetlanych informacji o drzewie. Jednak niektóre nazwy, na przykład `__init__`, są definiowane przez użytkownika i wymagają specjalnego traktowania. Naturalnym kolejnym krokiem może być rysowanie drzewa w interfejsie graficznym. Dostarczany razem z Pythonem pakiet narzędzi `tkinter`, który wykorzystaliśmy w przykładach w poprzednich podrozdziałach, oferuje podstawowe, ale proste w użyciu

funkcjonalności. Inne pakiety są bogatsze, ale też bardziej skomplikowane. Więcej wskazówek na ten temat można znaleźć w opisie przypadku na końcu rozdziału 28.

Fizyczne drzewa a dziedziczenie: wykorzystanie MRO (zapowiedź)

W następnym rozdziale ponownie zajmiemy się modelem klas w nowym stylu, który modyfikuje kolejność przeszukiwania klas w szczególnym przypadku dziedziczenia wielokrotnego (diamentowego). Przeanalizujemy atrybut `class.__mro__` obiektu klasy w nowym stylu, tj. krotkę zawierającą informacje o kolejności przeszukiwania drzewa dziedziczenia, tzw. kolejności MRO w nowym stylu.

Obecnie klasa `ListTree` wyświetla informacje o *fizycznym kształcie* drzewa dziedziczenia i oczekuje, że użytkownik sam wynioskuje, skąd dziedziczony jest dany atrybut. Wprawdzie takie było założenie, jednak uniwersalny program do przeglądania obiektów może również wykorzystywać krotkę MRO do automatycznego kojarzenia atrybutu z klasą, po której został *odziedziczony*. Skanując kolejność MRO w nowym stylu (lub klasyczną kolejność DFLR) dla każdego atrybutu znajdującego się w wyniku zwracanym przez funkcję `dir`, można symulować przeszukiwanie drzewa dziedziczenia i wiązać atrybuty z ich obiektemi źródłowymi w obrazie fizycznego drzewa klas.

W rzeczywistości *napiszemy* kod ideowo bardzo bliski opisanemu w następnym rozdziale modułowi `mapattrs`. Ponownie wykorzystamy klasy testowe z tego rozdziału do zademonstrowania tego pomysłu, zatem warto zaczekać na epilog tej historii. Ten kod będzie można wykorzystać jako zamiennik metody `_attrnames` wyświetlającej fizyczne lokalizacje atrybutów lub jako jej uzupełnienie. Obie odmiany mogą dostarczać programistom przydatnych informacji. Takie podejście jest również jednym ze sposobów obsługi slotów, które są tematem następnej wskazówki.

Wirtualne dane: sloty, właściwości i nie tylko (zapowiedź)

Ponieważ opisane tutaj klasy `ListInstance` i `ListTree` skanują słowniki `__dict__` przestrzeni nazw, mogą być przyczyną pewnych subtelnych problemów projektowych. Niektóre nazwy w klasach, powiązane z danymi instancji, mogą nie być w niej przechowywane. Dotyczy to nazw będących tematem następnego rozdziału, tj. właściwości w nowym stylu, slotów, deskryptorów, jak również atrybutów dynamicznie wyliczanych we wszystkich klasach za pomocą narzędzi takich jak metoda `__getattr__`. Żadna z nazw tych „wirtualnych” atrybutów nie jest przechowywana w słowniku przestrzeni nazw instancji, dlatego nie są one wyświetlane jako części własnych danych instancji.

Między innymi sloty wydają się najsilniej powiązane z instancją. Przechowują dane instancji, mimo że ich nazwy nie znajdują się w słownikach przestrzeni nazw. Właściwości i deskryptory również są powiązane z instancjami, jednak nie rezerwują w nich miejsca. Ich wyliczeniowa natura jest bardziej widoczna i są one bliższe metodom klasy niż danym instancji.

Jak zobaczymy w następnym rozdziale, sloty funkcjonują podobnie do atrybutów instancji, jednak są tworzone i zarządzane przez elementy automatycznie tworzone w klasach. Stanowią stosunkowo rzadko używaną funkcjonalność klas w nowym stylu. Są to atrybuty instancji deklarowane w atrubucie `__slots__` klasy i nie są fizycznie przechowywane w słowniku `__dict__` instancji. W rzeczywistości sloty mogą całkowicie zastępować słownik `__dict__`. Z tego powodu narzędzia, które skanują jedynie przestrzenie nazw instancji w celu wyświetlenia informacji o nich, nie są w stanie bezpośrednio wiązać instancji z atrybutami zapisanymi w slotach. Klasa `ListTree` w swojej oryginalnej postaci wyświetla sloty jako atrybuty klasy wszędzie tam, gdzie się pojawiają (jednak nie w instancji), a klasa `ListInstance` nie wyświetla ich w ogóle.

Powysze rozważania będą bardziej zrozumiałe po przeanalizowaniu tej funkcjonalności w następnym rozdziale, ale mają ona znaczenie dla prezentowanego tutaj kodu i podobnych narzędzi. Jeżeli na przykład w pliku `textmixin.py` umieścimy przypisanie `__slots__ = ['data1']` w klasie `Super`, a przypisanie `__slots__ = ['data3']` w klasie `Sub`, wtedy

wyświetlony zostanie tylko atrybut `data2` instancji obu powyższych klas. Klasa `ListTree` wyświetli również atrybuty `data1` i `data3`, ale jako atrybuty obiektów *klas Super* i *Sub*. Wyświetli też w specjalnym formacie ich wartości (z technicznego punktu widzenia są to deskryptory na poziomie klasy stanowiące opisane w następnym rozdziale inne narzędzia w nowym stylu).

Jak to zostanie wyjaśnione w następnym rozdziale, aby wyświetlić atrybuty slotów jako nazwy instancji, należy za pomocą metody `dir` uzyskać listę wszystkich atrybutów, zarówno fizycznych, jak i odziedziczonych. Następnie trzeba pobrać ich wartości z instancji za pomocą metody `getattr` lub ze źródła dziedziczenia za pomocą słownika `_dict_` i zaakceptować w klasach prezentację implementacji niektórych z nich. Ponieważ wynik metody `dir` zawiera nazwy odziedziczonych „wirtualnych” atrybutów, włącznie ze slotami i właściwościami, można je dołączyć do zbioru instancji. Jak się przekonamy, można tu wykorzystać algorytm MRO do wiązania atrybutów z ich źródłami, jak również filtrując nazwy odziedziczone po wbudowanej klasie `object`, można ograniczać wyświetlane informacje o klasach do nazw zakodowanych w klasach zdefiniowanych przez użytkownika.

Większość opisanych zmian nie dotyczy klasy `ListInherited`, ponieważ wyświetla ona pełny wynik metody `dir`, zawierający zarówno nazwy zawarte w słowniku `_dict_`, jak również w atrybutach `_slots_` wszystkich klas. Jednak przydatność informacji prezentowanych w obecnej formie jest niewielka. Do wyświetlania slotów można byłoby użyć odmiany klasy `ListTree` wykorzystującej metodę `dir` i kolejność MRO do powiązania atrybutów z klasami, ponieważ nazwy oparte na slotach pojawiają się jako narzędzia do zarządzania slotami w atrybutach `_dict_` klas, a nie instancji.

Innym rozwiązaniem jest przetwarzanie atrybutów opartych na slotach w obecny sposób bez komplikowania kodu na potrzeby obsługi rzadkich, zaawansowanych i kwestionowanych dzisiaj funkcjonalności. Sloty i zwykłe atrybuty instancji to nazwy różnych rodzajów. W rzeczywistości bardziej ścisłe jest wyświetlanie nazw slotów jako atrybutów klas, a nie instancji. Jak się przekonamy w następnym rozdziale, sloty implementuje się w klasach, choć ich miejsce jest w instancjach.

Jednak próbę zebrania wszystkich „wirtualnych” atrybutów powiązanych z klasami można potraktować jako pobożne życzenie. Techniki takie jak tutaj opisane można wykorzystać do obsługi slotów i właściwości. Jednak niektóre atrybuty są całkowicie dynamiczne, pozbawione jakichkolwiek fizycznych podstaw. Są to atrybuty wyliczane podczas pobierania ich za pomocą generycznych metod, takich jak `getattr`, i nie są to dane w zwykłym znaczeniu tego słowa. Narzędzia wyświetlające dane w tak dynamicznym języku jak Python muszą wyświetlać zastrzeżenie, że niektóre dane są *co najmniej ułotne!*^[3]

Na końcu tej części książki dokonamy niewielkiego rozszerzenia kodu opisanego w tym rozdziale. Na początku instancji będą wyświetlane w nawiasach nazwy klas nadzędnych. Dlatego warto zachować kod do wykorzystania w przyszłości. Aby zrozumienie dwóch ostatnich z powyższych wskazówek było jak najbardziej dokładne, musimy podsumować ten rozdział i przejść do następnej, ostatniej części książki.

Inne zagadnienia związane z projektowaniem

W tym rozdziale poznaliśmy wiele podstawowych zagadnień programowania zorientowanego obiektowo w Pythonie: dziedziczenie, kompozycję, delegację, wielokrotne dziedziczenie, metody związane i fabryki. Jednak jeśli chodzi o zagadnienia związane z projektowaniem zorientowanym obiektowo, to ledwie wierzchołek góry lodowej. Innych zagadnień przydatnych w projektowaniu warto szukać w pozostałych rozdziałach książki:

- klasy abstrakcyjne (rozdział 29.),
- dekoratory (rozdziały 32. i 39.),
- klasy typów (rozdział 32.),
- metody statyczne i metody klas (rozdział 32.),
- atrybuty zarządzane (rozdziały 32. i 38.),
- metaklasy (rozdziały 32. i 40.).

Jeśli jednak chodzi o zagadnienia ogólne ściślej związane z samym projektowaniem zorientowanym obiektowo, polecam literaturę poświęconą tej tematyce. Wzorce projektowe są bardzo ważnym elementem programowania zorientowanego obiektowo i w Pythonie ich stosowanie bywa często bardziej naturalne niż w innych językach, lecz nie są one dla Pythona specyficzne, dlatego najlepszym sposobem ich poznania jest doświadczenie.

Podsumowanie rozdziału

W niniejszym rozdziale zamieściliśmy przykłady często stosowanych sposobów wykorzystywania oraz łączenia klas w celu zoptymalizowania możliwości ich ponownego użycia i faktoryzacji. Często jest to uznawane za zagadnienia z zakresu projektowania, które są niezależne od języka programowania (choć Python może uprościć ich implementację). Omawialiśmy *delegację* (opakowanie obiektów w klasy pośredniczące), *kompozycję* (kontrolowanie obiektów osadzonych), *dziedziczenie* (nabywanie zachowania z innych klas), a także pewne bardziej ezoteryczne koncepcje, takie jak atrybuty pseudoprivatne, dziedziczenie wielokrotne, metody z wiązaniem oraz fabryki.

W kolejnym rozdziale zakończymy naszą pracę z klasami i programowaniem zorientowanym obiektowo, badając bardziej zaawansowane zagadnienia związane z klasami. Część tego materiału może być bardziej interesująca dla twórców narzędzi niż programistów aplikacji, jednak mimo to zasługuje na przejrzenie przez osoby, które chcą się zajmować programowaniem zorientowanym obiektowo w Pythonie — jeżeli nie we własnym kodzie, to w kodzie napisanym przez innych programistów, który trzeba zrozumieć. Najpierw jednak pora na krótki quiz.

Sprawdź swoją wiedzę — quiz

1. Czym jest dziedziczenie wielokrotne?
2. Czym jest delegacja?
3. Czym jest kompozycja?
4. Czym są metody związane?
5. Do czego wykorzystywane są atrybuty pseudoprivatne?

Sprawdź swoją wiedzę — odpowiedzi

1. Dziedziczenie wielokrotne ma miejsce, kiedy klasa dziedziczy po więcej niż jednej klasie nadzędnej. Przydaje się to do mieszania ze sobą kilku pakietów kodu

opartego na klasach. Przy wyszukiwaniu atrybutów obowiązuje kolejność przeszukiwania nazw od lewej do prawej względem kolejności deklaracji dziedziczenia.

2. Delegacja obejmuje opakowanie obiektu w klasę pośredniczącą, dodającą dodatkowe zachowanie i przekazującą inne operacje do opakowanego obiektu. Klasa pośrednicząca (proxy) zachowuje interfejs opakowanego obiektu.
3. Kompozycja jest techniką, w której klasa kontrolera osadza i kieruje kilkoma obiektami, a także udostępnia własny interfejs. Jest to sposób tworzenia większych struktur za pomocą klas.
4. Metody z wiązaniem łączą instancję oraz funkcję metody. Można je wywoływać bez przekazywania obiektu instancji w sposób jawnym, ponieważ oryginalna instancja nadal jest dostępna.
5. Atrybuty pseudoprывatne (`__X`, czyli takie, których nazwy rozpoczynają się, ale nie kończą dwoma znakami podkreśnika) są wykorzystywane do ograniczania zasięgu nazw do klasy, w której są zdefiniowane. Dotyczy to atrybutów klasy, jak zdefiniowane w niej metody, oraz atrybutów instancji `self`, przypisanych w ramach klasy. Nazwy te są przekształcane w taki sposób, aby zawierały przedrostek nazwy klasy, co powoduje, że stają się unikalne.

[1] Wydaje się to niepotrzebnie niepokoić programistów języka C++. W Pythonie można nawet zmodyfikować lub całkowicie usunąć metodę klasy w czasie wykonywania. Z drugiej strony prawie nikt tego nie robi w prawdziwych programach. Jako język skryptowy Python bardziej podkreśla umożliwianie niż ograniczanie. Można sobie przypomnieć, że w omówieniu przeciążania operatorów w rozdziale 30. wspomnieliśmy, iż metody `__getattr__` oraz `__setattr__` można wykorzystać do emulacji prywatności. Zazwyczaj nie są one jednak używane w tym celu w praktyce. Wróćmy do tego tematu przy okazji tworzenia bardziej realistycznego dekoratora prywatności w rozdziale 39.

[2] Tak naprawdę taka składnia może posłużyć do wywołania dowolnego obiektu wywoływalnego, w tym funkcje, klasy oraz metody. W taki sposób funkcja `factory` może także wykonać dowolny obiekt wywoływalny, nie tylko klasę (pomimo nazwy argumentu). W rozdziale 18. poznaliśmy też dostępną w Pythonie 2.x alternatywę dla wywołania `aClass(*pargs, **kargs)`: funkcję wbudowaną `apply(aClass, pargs, kargs)`, która w Pythonie 3.x została usunięta z powodu nadmiarowości i ograniczeń.

[3] Niektóre dynamiczne i pośredniczące obiekty oparte na metodzie `__getattr__` i podobnych również mogą wykorzystywać metodę przeciążającą operator `dir`, by publikować listę atrybutów dla wywołań `dir`. Ponieważ jest to opcjonalne rozwiązanie, uniwersalne narzędzia nie mogą polegać na klasach ich klientów, aby wykonać tę operację. Więcej informacji na temat metody `__dir__` zawiera piąte wydanie książki *Python. Leksykon kieszonkowy*.

Rozdział 32. Zaawansowane zagadnienia związane z klasami

Niniejszy rozdział kończy nasze omówienie programowania zorientowanego obiektowo, prezentując kilka bardziej zaawansowanych zagadnień związanych z klasami. Omówimy tworzenie klas podzielonych dla typów wbudowanych, zmiany i rozszerzenia w klasach w nowym stylu, metody statyczne, metody klasy, sloty i właściwości, dekoratory funkcji i klas, algorytm MRO, wywołanie super i wiele innych.

Jak widzieliśmy, model programowania zorientowanego obiektowo w Pythonie jest bardzo prosty, a część zagadnień zaprezentowanych w niniejszym rozdziale jest na tyle zaawansowana i opcjonalna, że możemy nie spotkać się z nimi zbyt często w naszej karierze twórcy aplikacji Pythona. W trosce o kompletność materiału, jak również dlatego, że nigdy nie wiadomo, kiedy „zaawansowane” zagadnienie może przydać się w kodzie, zakończymy nasze omawianie klas krótkim spojrzeniem na bardziej skomplikowane narzędzia przeznaczone do zaawansowanego programowania zorientowanego obiektowo.

Ponieważ jest to ostatni rozdział tej części książki, jak zawsze zakończymy go podrozdziałem dotyczącym pułapek związanych z klasami, a także częścią z ćwiczeniami dotyczącymi omawianych w tej części książki zagadnień. Zachęcam do wykonania tych ćwiczeń w celu utrwalenia zawartego tu materiału. Sugeruję również pracę lub przestudiowanie większych projektów opartych na programowaniu zorientowanym obiektowo w Pythonie w celu uzupełnienia podanych w książce informacji. Jak zawsze w przypadku informatyki zalety programowania zorientowanego obiektowo stają się widoczne wraz z wykonywaniem praktycznych zadań.



Uwaga na temat treści: Ten rozdział zawiera opisy bardziej zaawansowanych zagadnień, jednak część z nich jest zbyt skomplikowana, aby je tu solidnie omówić. Takie tematy, jak właściwości, deskryptory, dekoratory i metaklasy są tu ledwie wspomniane, a ich pełna analiza znalazła się w końcowej części książki, po rozdziale o wyjątkach. Umieszczone są tam bardziej rozbudowane przykłady i rozszerzony materiał poświęcone tematom poruszonym w tym rozdziale.

Należy pamiętać, że niniejszy rozdział jest najdłuższy ze wszystkich. Zakładam, że czytelnik ma dość odwagi i jest gotów zakazać rękawy, aby zmierzyć się z opisanymi tu tematami i dogłębiście je zbadać. Jeżeli jednak zagadnienia z dziedziny programowania obiektowego nie będą interesujące, może przejść do materiału w końcowej części rozdziału i wrócić tutaj w przyszłości, gdy będzie potrzebował zastosować prezentowane tu narzędzia w swoim kodzie.

Rozszerzanie typów wbudowanych

Poza implementowaniem nowych obiektów klasy są czasami wykorzystywane do rozszerzania funkcjonalności typów wbudowanych Pythona w celu dodania obsługi bardziej egzotycznych struktur danych. By na przykład dodać do list metody wstawiania oraz usuwania kolejek, możemy napisać kod klas opakowujących (osadzających) obiekty listy i eksportować metody

wstawiania oraz usuwania przetwarzające listę w specjalny sposób, podobnie jak technika delegacji omówiona w rozdziale 31. Od Pythona 2.2 można również wykorzystać dziedziczenie do specjalizowania typów wbudowanych. W kolejnych dwóch podrozdziałach zaprezentujemy działanie obu tych technik.

Rozszerzanie typów za pomocą osadzania

Przypomnijmy sobie funkcje działające na zbiorach, które napisaliśmy w rozdziałach 16. i 18. książki. Oto jak wyglądają, kiedy się je przywróci do życia w postaci klas Pythona. Poniższy przykład (*setwrapper.py*) implementuje nowy typ obiektu zbioru, przenosząc część funkcji zbiorów do metod i dodając pewne podstawowe metody przeciążania operatorów. Klasa ta w dużej mierze opakowuje po prostu listę Pythona za pomocą dodatkowych operacji na zbiorach. Ponieważ jest klasą, obsługuje również wiele instancji oraz możliwość dostosowania do własnych potrzeb za pomocą dziedziczenia w klasach podrzędnych. W odróżnieniu od funkcji prezentowanych wcześniej, do których były przekazywane listy wartości, klasy użyte w tym przykładzie tworzą obiekty typu zbiorowego o zadanych z góry danych i zachowaniu.

```
class Set:
    def __init__(self, value = []):                      # Konstruktor
        self.data = []                                     # Zarządza listą
        self.concat(value)

    def intersect(self, other):                          # other to dowolna sekwencja
        res = []                                         # self to podmiot
        for x in self.data:
            if x in other:                               # Wybór wspólnych elementów
                res.append(x)
        return Set(res)                                 # Zwrócenie nowej instancji Set

    def union(self, other):                            # other to dowolna sekwencja
        res = self.data[:]                            # Kopia mojej listy
        for x in other:                               # Dodanie elementów w other
            if not x in res:
                res.append(x)
        return Set(res)

    def concat(self, value):                           # value: lista, Set...
        for x in value:                             # Usuwa duplikaty
            if not x in self.data:
                self.data.append(x)

    def __len__(self):                                return len(self.data)          #
    len(self)

    def __getitem__(self, key):                         return self.data[key]         #
    self[i], self[i:j]
```

```

        def __and__(self, other):           return self.intersect(other) #
self & other

        def __or__(self, other):          return self.union(other)      #
self | other

        def __repr__(self):             return 'Zbiór:' + repr(self.data) #
Wyświetlenie za

# pomocą print

        def __iter__(self):            return iter(self.data)         #
for x in self, ...

```

Aby użyć tej klasy należy w zwykły sposób utworzyć jej instancję, wywołać metody i zastosować zdefiniowane operatory.

```

from setwrapper import Set
x = Set([1, 3, 5, 7])
print(x.union(Set([1, 4, 7])))      # Wypisuje Set:[1, 3, 5, 7, 4]
print(x | Set([1, 4, 6]))          # Wypisuje Set:[1, 3, 5, 7, 4, 6]

```

Przeciążając operatory indeksowania i iterowania, można za pomocą instancji naszej klasy `Set` symulować prawdziwe listy. Ponieważ nad tą klasą i jej rozszerzeniem będziemy pracować w ćwiczeniu na końcu rozdziału, nie powiem nic więcej na temat kodu aż do dodatku D.

Rozszerzanie typów za pomocą klas podrzędnych

Od Pythona 2.2 wszystkie typy wbudowane można bezpośrednio rozszerzać za pomocą klas podrzędnych. Funkcje konwersji typów, takie jak `list`, `str`, `dict` oraz `tuple`, stały się nazwami typów wbudowanych. Choć jest to niewidoczne dla naszego skryptu, wywołanie konwersji typu (na przykład `list('mielonka')`) jest teraz tak naprawdę wywołaniem konstruktora obiektu typu.

Zmiana ta pozwala rozszerzać lub dostosowywać do własnych potrzeb zachowanie typów wbudowanych za pomocą zdefiniowanych przez użytkownika instrukcji `class`. By dostosować typ wbudowany do własnych potrzeb, wystarczy utworzyć klasę podzielną z wykorzystaniem nowej nazwy typu. Instancje klas nadziedznych typów można wykorzystywać w każdym miejscu, w którym mógłby się pojawić oryginalny typ wbudowany. Założymy na przykład, że mamy problem z przyzwyczajeniem się do faktu, iż wartości przesunięcia w indeksach list Pythona zaczynają się od 0, a nie od 1. Nie ma się co martwić — zawsze możemy utworzyć własną klasę dostosowującą tę właściwość list do naszych upodobań. Plik `typesubclass.py` pokazuje, jak można to zrobić.

```

# Klasa podzielna wbudowanego typu (klasy) listy
# Odwzorowanie 1..N na 0..N-1; wywołanie z powrotem wbudowanej wersji
class MyList(list):

    def __getitem__(self, offset):
        print('(indeksowanie %s w pozycji %s)' % (self, offset))
        return list.__getitem__(self, offset - 1)

    if __name__ == '__main__':

```

```

print(list('abc'))

x = MyList('abc')                      # Metoda __init__ odziedziczona po liście
                                         # Metoda __repr__ odziedziczona po liście
print(x)
                                         # MyList.__getitem__
print(x[1])
                                         # Dostosowuje metodę klasy nadzędnej
print(x[3])
                                         # Dostosowuje metodę klasy nadzędnej
listy

x.append('mielonka'); print x    # Atrybuty z klasy nadzędnej listy
x.reverse(); print x

```

W powyższym pliku klasa podrzędna `MyList` rozszerza metodę indeksowania listy wbudowanej `__getitem__` jedynie w taki sposób, by odwzorować indeksy od 1 do N na wymagane 0 do N-1. Tak naprawdę zmniejsza wartość otrzymanego indeksu i wywołuje z powrotem wersję indeksowania z klasy nadzędnej, jednak w zupełności wystarczy to do wykonania tego zadania.

```

% python typesubclass.py

['a', 'b', 'c']
['a', 'b', 'c']
(indeksowanie ['a', 'b', 'c'] w pozycji 1)
a
(indeksowanie ['a', 'b', 'c'] w pozycji 3)
c
['a', 'b', 'c', 'mielonka']
['mielonka', 'c', 'b', 'a']

```

Powyższe dane wyjściowe obejmują również tekst śledzenia, który klasa wyświetla przy indeksowaniu. Czy taka modyfikacja indeksowania jest dobrym pomysłem to inna sprawa — użytkowników klasy `MyList` może bardzo zdziwić takie odejście od zachowania sekwencji Pythona. To, że możemy dostosować typy wbudowane w ten sposób, daje nam jednak narzędzie o dużych możliwościach.

Powyższy wzorzec kodu umożliwia nam na przykład alternatywne sposoby tworzenia zbiorów, które mogą być klasą podrzędną wbudowanego typu `list` zamiast samodzielna klasą zarządzającą osadzonym obiektem listy. Jak dowiedzieliśmy się w rozdziale 5., Python oferuje wbudowany typ zbiorowy oraz składnię definiowania literałów i złożień, pozwalającą na tworzenie nowych zbiorów. Jednak samodzielna implementacja klasy implementującej `zbiory` to nadal doskonały sposób na nauczenie się technik tworzenia klas potomnych.

Poniższa klasa utworzona w pliku `setsubclass.py` dostosowuje listy w taki sposób, by dodać do nich jedynie metody oraz operatory powiązane z przetwarzaniem zbiorów. Ponieważ pozostałe zachowanie dziedziczone jest po wbudowanej klasie nadzędnej `list`, umożliwia nam to utworzenie krótszej i prostszej alternatywy. Wszystkie inne wywołania metod są kierowane bezpośrednio do klasy `list`:

```

from __future__ import print_function    # Kompatybilność z wersją 2.x
class Set(list):
    def __init__(self, value = []):        # Konstruktor
        list.__init__(self)                 # Dostosowuje listę do własnych
potrzeb

```

```

        self.concat(value)                      # Kopiuje zmienne wartości domyślne

def intersect(self, other):                  # other to dowolna sekwencja

    res = []
                                # self to podmiot

    for x in self:
        if x in other:                      # Wybór wspólnych elementów
            res.append(x)

    return Set(res)                      # Zwrócenie nowej instancji Set

def union(self, other):                    # other to dowolna sekwencja

    res = Set(self)
                                # Kopia mojej listy

    res.concat(other)

    return res

def concat(self, value):                  # value: lista, Set...

    for x in value:
        if not x in self:
            self.append(x)

def __and__(self, other): return self.intersect(other)

def __or__(self, other): return self.union(other)

def __repr__(self): return 'Zbiór:' + list.__repr__(self)

if __name__ == '__main__':
    x = Set([1,3,5,7])
    y = Set([2,1,4,5,6])
    print(x, y, len(x))
    print(x.intersect(y), y.union(x))
    print(x & y, x | y)
    x.reverse(); print(x)

```

Poniżej znajdują się dane wyjściowe kodu testującego znajdującego się na końcu pliku. Ponieważ tworzenie klas podlegających dla typów wbudowanych jest zaawansowaną opcją, przeznaczoną dla ograniczonej grupy użytkowników, pominię dalsze szczegóły, jednak zapraszam do prześledzenia wyników w celu przestudiowania tego zachowania kodu (takiego samego w wersjach 2.x i 3.x).

```
% python setsubclass.py
Zbiór:[1, 3, 5, 7], Zbiór:[2, 1, 4, 5, 6], 4
Zbiór:[1, 5], Zbiór:[2, 1, 4, 5, 6, 3, 7]
Zbiór:[1, 5], Zbiór:[1, 3, 5, 7, 2, 4, 6]
Zbiór:[7, 5, 3, 1]
```

Istnieją bardziej wydajne sposoby implementowania zbiorów w Pythonie za pomocą słowników, zastępujące liniowe przeszukiwanie w pokazanych wyżej implementacjach zbiorów operacją indeksowania słowników (mieszania), co działa o wiele szybciej. Więcej informacji na ten temat można znaleźć w książce *Programming Python*. Osoby zainteresowane zbiorami mogą również przyjrzeć się wbudowanemu typowi obiektu `set`, który omawialiśmy w rozdziale 5. Typ ten udostępnia działania na zbiorach jako narzędzia wbudowane. Implementacje zbiorów są zabawnym eksperymentem, jednak nie są one w dzisiejszym Pythonie wymagane.

Kolejny przykład klas podrzędnych dla typów można zobaczyć na podstawie nowego typu `bool` w Pythonie 2.3 oraz późniejszej wersji. Jak wspomnieliśmy wcześniej, `bool` jest klasą podrzędną typu `int` mającą dwie instancje (`True` oraz `False`), które zachowują się jak liczby całkowite 1 oraz 0, jednak dziedziczą własne metody reprezentacjiłańcuchów znaków, które wyświetlają ich nazwy.

Klasy w nowym stylu

W Pythonie 2.2 wprowadzono nowy rodzaj klas, zwany klasami „w nowym stylu”. Klassy omawiane dotychczas w książce są nazywane „klasycznymi”, kiedy porównuje się je z nowym typem. W Pythonie 3.x migracja do klas w nowym stylu została zakończona, ale w przypadku wersji Pythona z serii 2.x nadal mamy do czynienia z podziałem na klasyczny model klas i model klas w nowym stylu.

- Od *Pythona 3.x* wszystkie klasysą automatycznie tworzone jako klasys w nowym stylu, niezależnie od tego, czy dziedziczą po klasie `object`, czy nie. Kodowanie dziedziczenia klasie `object` nie jest obowiązkowe, ponieważ jest wykonywane niejawnie.
- W *Pythonie 2.x* i wcześniejszych klasys w nowym stylu muszą jawnie dziedziczyć po klasie `object` (lub po innym typie wbudowanym). Wszystkie inne klasysą „klasyczne”.

Dzięki temu, że w 3.x wszystkie klasysą automatycznie klasami w nowym stylu, opisywane przez nas funkcje, szczególnie dla klas w nowym stylu, są normą. Zdecydowałem się jednak, że te cechy opiszę osobno — z myślą o użytkownikach Pythona w wersjach 2.x — aby zaznaczyć, że w celu ich udostępnienia klasom należy zadeklarować dziedziczenie po klasie `object`.

Innymi słowy, użytkownicy Pythona 3.x, widząc w tej książce temat „klas w nowym stylu”, powinni traktować go jako odnoszący się do istniejących właściwości utworzonych przez nich klas. Dla użytkowników Pythona 2.x są to opcjonalne rozszerzenia i zmiany, które można wprowadzić, ale nie trzeba tego robić, chyba że są zastosowane w wykorzystywanym kodzie.

W Pythonie 2.x i wcześniejszych jedyna *składniowa* różnica deklaracji klas w nowym stylu polega na tym, że deklaruje się jej klasę nadczną w postaci typu wbudowanego (na przykład `list`) lub specjalnego typu `object`. Nowa nazwa wbudowana `object` udostępniona została w taki sposób, by mogła służyć jako klasa nadczenna dla klas w nowym stylu, jeśli żaden inny typ wbudowany nie nadaje się do zastosowania w określonym przypadku.

```
class newstyle(object):          # Jawnia deklaracja klasys w nowym stylu w
    wersji 2.x,
    ...normalny kod...           # wykonywana automatycznie, niewymagana w
    wersji 3.x
```

Każda klasa pochodząca od `object` lub innego typu wbudowanego jest automatycznie traktowana jak klasa w nowym stylu. Dopóki typ wbudowany znajduje w którejś klasie nadczennej, nowa klasa w wersji 2.x przejmuje jej działanie i rozszerzenia. Klasys niepochodzące od typów wbudowanych uznawane są za klasyczne.

Jak nowy jest nowy styl

Jak się przekonamy, zmiany wprowadzone w klasach w nowym stylu, szczególnie jeżeli wykorzystywane są ich zaawansowane funkcjonalności, są bardzo głębokie i mają istotny wpływ na działanie kodu. W rzeczywistości przekształcają one Pythona w *nowy język programowania obiektowego* — obowiązkowy w wersji 3.x i opcjonalny w 2.x, przejmujący wiele cech (często również skomplikowanych) z innych języków.

Nowy styl klas jest częściowo próbą połączenia pojęcia *klasa* z pojęciem *typ* istniejącym mniej więcej od wersji Pythona 2.2. Wielu użytkowników nawet tej zmiany nie zauważało, dopóki nie stała się wymagana w wersji 3.x. Sukces tej zmiany należy oceniać indywidualnie, jednak moim zdaniem w tym modelu różnice nadal istnieją, tym razem pomiędzy pojęciami *klasa* a *metaklasa*. Teraz zwykłe klasy są bardziej użyteczne, jednak efektem ubocznym jest ich znacznie większa złożoność. Na przykład opisany w rozdziale 40. algorytm dziedziczenia klas w nowym stylu jest co najmniej dwa razy bardziej skomplikowany niż poprzednio.

Niemniej jednak programiści tworzący prosty kod zauważają jedynie drobne różnice w porównaniu z tradycyjnymi klasami. Ostatecznie w prezentowanych dotąd rozbudowanych przykładach klas wprowadzona zmiana była jedynie sygnalizowana. Ponadto model klas istniejący w wersji 2.x funkcjonuje niezmiennie tak samo pod prawie dwóch dekad[1].

Ponieważ jednak zasadniczy sposób zmienił się funkcjonowanie klas, nowy styl musiał zostać w wersji 2.x wprowadzony jako osobne narzędzie, aby nie wpływał negatywnie na istniejący kod, który oparty był na poprzednim modelu. Na przykład subtelne różnice w algorytmie wyszukiwania nazw w tzw. diamentowym wzorcu dziedziczenia oraz mechanizm wyszukiwania metod specjalnych dla operacji wbudowanych, jak `__getattr__`, mogą spowodować, że istniejący kod przestanie działać, jeżeli nie wprowadzi się w nim zmian. Podobny efekt może mieć zastosowanie w nowym modelu opcjonalnych rozszerzeń, na przykład slotów.

Podział pomiędzy modelami klas został zniesiony w wersji 3.x, w której nowy styl jest *obowiązkowy*. Wciąż jednak istnieje w wersji 2.x i w nowym kodzie oraz istniejącym produkcyjnym, wykorzystującym poprzednią wersję. Ponieważ nowy model jest opcjonalnym rozszerzeniem wersji 2.x, w kodzie zgodnym z tą wersją można stosować jeden z wybranych modeli klas.

Następne dwa punkty zawierają przegląd nowości wprowadzonych w klasach w nowym stylu. Oczywiście różnice mają znaczenie dla użytkowników Pythona 2.x, natomiast ci, którzy na co dzień pracują z Pythonem 3.x, mogą ten opis potraktować jako listę zaawansowanych właściwości klas. Znajdą oni tutaj pełny opis modelu, po części prezentowanego w kontekście zmian, które można traktować jak funkcjonalności, ale tylko wtedy, gdy nie trzeba zajmować się milionami wierszy kodu napisanego w wersji 2.x.

Nowości w klasach w nowym stylu

Klasy w nowym stylu różnią się od klasycznych pod kilkoma względami, niektóre z nich są subtelne, ale mogą powodować niezgodność z istniejącym kodem Pythona 2.x. Poniżej omawiamy najistotniejsze z tych różnic.

Odczytywanie atrybutów w operacjach wbudowanych: pomijanie instancji

Podstawowe metody `__getattribute__` i `__getattr__` są dalej wywoływane podczas jawnego odwoływanego się do atrybutów za pomocą ich nazw, w przeciwieństwie do niejawnego odwoływanego się podczas wykonywania operacji wbudowanych. Nie są wywoływane jedynie w przypadku wbudowanych metod `X` przeciążających operatory. Wyszukiwanie tych metod odbywa się w klasach, nie w instancjach. Uniemożliwia to lub

komplikuje działanie obiektów pełniących role pośredników w interfejsach innych obiektów, jeżeli w obiektach opakowanych zaimplementowane jest przeciążanie operatorów. Aby tego rodzaju metody były rozróżniane podczas kierowania wbudowanych operacji w klasach w nowym stylu, należy ponownie je zdefiniować.

Połączenie klas i typów: sprawdzanie typu

Klasy są teraz typami, a typy są klasami. W rzeczywistości oba pojęcia stały się synonimami, choć *metaklasa* zastępująca *typ* różni się od zwykłej klasy. Funkcja wbudowana *type(I)* zwróci klasę, z której utworzono obiekt *I*, nie generyczny typ, i najczęściej jest to ta sama klasa, którą zawiera *I.__class__*. Co więcej, klasy są instancjami klasy *type*; można również tworzyć klasy potomne klasy *type*, co pozwala na dostosowanie do własnych potrzeb mechanizmu tworzenia klas za pomocą instrukcji *class*. Może to mieć znaczenie w kodzie, który sprawdza typy danych lub w inny sposób opiera się na poprzednim modelu typów.

Domyślna klasa główna object

Wszystkie klasy w nowym stylu (czyli typy) pochodzą od klasy *object* zawierającej niewielki zestaw metod przeciążających operatory (np. *__repr__*). W wersji 3.x klasa ta jest dodawana automatycznie ponad zdefiniowaną przez użytkownika klasą główną na najwyższym poziomie hierarchii i nie trzeba jej jawnie wskazywać jako klasy nadzędnej. Może to mieć wpływ na kod, w którym zakłada się brak domyślnych metod i klas głównych.

Porządek przeszukiwania drzewa dziedziczenia: algorytm MRO i diamentowa struktura drzewa

Kolejność wyszukiwania nazw w diamentowym schemacie wielokrotnego dziedziczenia jest teraz nieco inna: przeszukiwanie najpierw odbywa się w poprzek, a następnie wzdłuż drzewa. Proces wyszukiwania atrybutów, tj. algorytm MRO, można śledzić za pomocą atrybutu *__mro__* dostępnego w klasach w nowym stylu. Nowy porządek dotyczy głównie tylko diamentowych drzew klas, choć domniemany główny obiekt w nowym modelu tworzy diament we wszystkich drzewach wielokrotnego dziedziczenia. Kod opierający się na poprzednim porządku przeszukiwania nie będzie działał w taki sam sposób jak do tej pory.

Algorytm dziedziczenia: rozdział 40.

Algorytm dziedziczenia klas w nowym stylu jest znacznie bardziej skomplikowany niż w przypadku typowych klas z porządkiem wyszukiwania nazw wzdłuż drzewa. Wykorzystywane są w nim specjalne deskryptory, metaklasy i wbudowane operacje. Algorytm ten zostanie sformalizowany dopiero w rozdziale 40. szczegółowo opisującym metaklasy i deskryptory. Nowy algorytm może mieć wpływ na kod, który nie uwzględnia tego rodzaju zawiłości.

Nowe zaawansowane narzędzia: wpływ na kod

Klasy w nowym stylu oferują zestaw nowych narzędzi, jak *sloty*, *właściwości*, *deskryptory*, funkcję *super* i metodę *__getattribute__*. Większość z tych narzędzi ma bardzo specjalizowane zastosowania, przede wszystkim do konstruowania narzędzi programistycznych. Ich użycie może jednak zakłócić lub uniemożliwić działanie istniejącego kodu. Na przykład sloty niekiedy całkowicie uniemożliwiają tworzenie słownika przestrzeni nazw instancji, a podstawowe metody obsługi atrybutów mogą wymagać innego zakodowania.

Rozszerzenia wymienione w dwóch ostatnich grupach będą opisane w osobnym podrozdziale, natomiast opis algorytmu dziedziczenia jest odłożony do rozdziału 40. Ponieważ inne wymienione wyżej różnice mogą potencjalnie uniemożliwić działanie tradycyjnego kodu, przyjrzyjmy się im dokładniej teraz.

Uwaga na temat treści: należy pamiętać, że zmiana polegająca na wprowadzeniu klas w nowym stylu dotyczy wersji 3.x i 2.x, mimo że w tej ostatniej jest to opcja. W tym rozdziale i w innych miejscach książki zmiana ta



jest nazywana funkcjonalnością wersji 3.x, aby podkreślić jej odmiennosć w stosunku do wersji 2.x. Stosowanie klas w nowym stylu jest obowiązkowe w wersji 3.x, natomiast w wersji 2.x również można ich używać. Różnica ta jest często podkreślana, ale nie dogmatycznie. Aby rzecz bardziej skomplikować, zmiany dotyczące klas w wersji 3.x wynikają z wprowadzenia klas w nowym stylu (np. pomijanie metody `__getattr__` w metodach operatorów), jednak nie dotyczy to wszystkich (na przykład zastąpienia niepowiązanych metod funkcjami). Co więcej, wielu programistów używających wersji 2.x pozostaje przy typowych klasach, ponieważ traktują je jako funkcjonalność wersji 3.x. Klasa w nowym stylu nie są jednak nowością i istnieją w obu wersjach Pythona. Skoro są obecne w wersji 2.x, informacje o nich są lekturą obowiązkową dla jej użytkowników.

Pomijanie instancji we wbudowanych operacjach przy pobieraniu atrybutów

Klasy w nowym stylu zostały wspomniane w uwagach w rozdziałach 28. i 31. ze względu na ich znaczenie w zawartych tam przykładach i zagadnieniach. W tego rodzaju klasach (czyli we wszystkich w wersji 3.x) podstawowe metody instancji przechwytyjące odwołania do atrybutów, tj. `__getattr__` i `__getattribute__`, nie są już wywoływanie przez przeciążone metody `X` wbudowanych operacji. Wyszukiwanie ich nazw rozpoczyna się w klasach, a nie w instancjach. Jednak jawne odwołania do atrybutów są kierowane do powyższych metod, mimo że mają one nazwy `X`. Jest to zasadnicza zmiana w realizacji operacji wbudowanych.

Mówiąc bardziej formalnie, jeżeli w klasie zdefiniowana jest metoda przeciążająca operację indeksowania `__getitem__`, a X jest instancją tej klasy, wtedy operacja indeksowania, np. `X[1]`, jest mniej więcej równoważna instrukcji `X.__getitem__(1)` w typowej klasie, natomiast jest ścisłym odpowiednikiem instrukcji `type(X).__getitem__(X, 1)` w klasie w nowym stylu. W tym drugim przypadku wyszukiwanie nazwy rozpoczyna się w klasie, a więc pomijana jest metoda `__getattr__` w instancji o niezdefiniowanej nazwie.

Z technicznego punktu widzenia wyszukiwanie metody wykonującej wbudowaną operację, np. `X[1]`, rozpoczyna się na poziomie klas, uwzględniane jest zwykle dziedziczenie i sprawdzane są tylko słowniki przestrzeni nazw wszystkich klas, od których pochodzi klasa X. Jest to zmiana, która może mieć znaczenie w modelu *metaklas* opisany w dalszej części rozdziału i w rozdziale 40., gdzie klasy mogą przejmować działanie na różne sposoby. Instancja jest jednak pomijana podczas wyszukiwania wbudowanych operacji.

Dlaczego zmieniło się wyszukiwanie?

Formalne powody wprowadzenia opisanej wyżej zmiany można znaleźć w różnych źródłach. W tej książce nie będę uzasadniał zmian potencjalnie uniemożliwiających działanie wielu poprawnych programów. Możemy przyjąć, że było to posunięcie *optymalizacyjne* i próba rozwiązania problemu z niejasnym wzorcem wywołań. Pierwszy powód wynika z częstotliwości stosowania wbudowanych operacji. Jeżeli na przykład przy każdym dodawaniu muszą być wykonywane dodatkowe operacje na instancji, wtedy program spowalnia działanie, szczególnie gdy uwzględniane są liczne rozszerzenia na poziomie atrybutów w modelu w nowym stylu.

Drugi powód jest mniej oczywisty i został opisany w dokumentacji do Pythona. W skrócie: odzwierciedla on zagadkę wprowadzoną przez model *metaklas*. Ponieważ klasa są instancjami metaklas, a w metaklasach można definiować metody wbudowanych operatorów przetwarzających generowane klasy, wywołanie metody klasy musi pomijać samą klasę i szukać na wyższym poziomie metody, która przetwarza tę klasę, zamiast wybierać wersję z własnej klasy. Wybranie własnej wersji spowodowałoby wywołanie niepowiązanej metody, ponieważ własna metoda klasy przetwarza niższe instancje. Jest to zwykły model niepowiązanych metod

opisany w poprzednim rozdziale, ale potencjalnie gorszy ze względu na fakt, że klasy mogą przejmować działania typów również od metaklas.

W efekcie, ponieważ klasy są zarówno typami, jak i pełnoprawnymi instancjami, wszystkie instancje są pomijane podczas wyszukiwania metod wbudowanych operacji. Z założenia dotyczy to zwykłych instancji, aby zapewnić jednolitość i spójność, ale instancje są sprawdzane zarówno przy odwołaniach do nazw operacji innych niż wbudowane, jak również przy bezpośrednich i jawnych odwołaniach do nazw wbudowanych. Choć jest to niewątpliwie konsekwencją wprowadzenia klas w nowym stylu, może wyglądać na rozwiązywanie problemu wzorca użytkowania, bardziej sztucznego i niejasnego niż szeroko stosowany obecnie. Droga ku optymalizacji wydaje się w ten sposób bardziej uzasadniona, choć nie jest wolna od ubocznych skutków.

Szczególnie istotne implikacje wynikają w przypadku klas *delegujących*, często zwanych klasami *pośredniczącymi*, gdy osadzony obiekt implementuje przeciążanie operatorów. W przypadku klas w nowym stylu w tego rodzaju obiekcie trzeba ponownie definiować wszystkie nazwy (ręcznie lub przy użyciu narzędzi), aby móc je przechwytywać i delegować. Ostatecznym efektem jest znaczne skomplikowanie albo całkowite wyeliminowanie *całej kategorii programów*. Delegowanie zostało opisane w rozdziałach 28. i 31. Jest to popularny wzorzec często stosowany do rozszerzania i dostosowywania interfejsu innej klasy, np. w celu zaimplementowania weryfikacji wartości, śledzenia działania, mierzenia czasu wykonania i wielu innych operacji. Choć stosowanie klas pośredniczących w typowym kodzie jest raczej wyjątkiem niż regułą, wiele programów się na nich opiera.

Implikacje wynikające z przechwytywania atrybutów

Aby dowiedzieć się, czym klasy w nowym stylu różnią się od zwykłych, najprościej będzie uruchomić kod w wersji 2.x, w którym operacje indeksowania i wyświetlania są kierowane w tradycyjnej klasie do metody `__getattr__`. Inaczej jest w klasie w nowym stylu, gdzie operacja wyświetlania jest wykonywana w domyślny sposób^[2]:

```
>>> class C:
        data = 'spam'

        def __getattr__(self, name):      # Klasyczne w wersji 2.x: przechwytuje
            wszystkie operacje
                                            # wbudowane

            print(name)

            return getattr(self.data, name)

>>> X = C()

>>> X[0]

__getitem__
's'

>>> print(X)                      # Zwykła klasa nie dziedziczy
domyślnych cech

__str__
spam

>>> class C(object):              # Nowy styl w wersjach 2.x i 3.x
...pozostała część kodu bez zmian...
```

```

>>> X = C()                                # Wbudowane operacje nie są kierowane
do setattr

>>> X[0]
TypeError: 'C' object does not support indexing

>>> print(X)
<__main__.C object at 0x02205780>

```

Tej rozbieżności, pozornie uzasadniającej nazwy metod metaklas i optymalizującej wbudowane operacje, nie rozwiązują zwykłe instancje zawierające metodę `__getattr__`. Dotyczy to wbudowanych operacji, a nie metod o zwykłych nazwach ani wbudowanych metod jawnie wywoływanych za pomocą ich nazw:

```

>>> class C: pass                         # Zwykła klasa w wersji 2.x

>>> X = C()

>>> X.normal = lambda: 99

>>> X.normal()
99

>>> X.__add__ = lambda(y): 88 + y

>>> X.__add__(1)
89

>>> X + 1
89

>>> class C(object): pass                  # Klasa w nowym stylu w wersji 2.x/3.x

>>> X = C()

>>> X.normal = lambda: 99

>>> X.normal()                           # Zwykła metoda instancji
99

>>> X.__add__ = lambda(y): 88 + y

>>> X.__add__(1)                          # Tak samo dla jawnych, wbudowanych
nazw
89

>>> X + 1
TypeError: unsupported operand type(s) for +: 'C' and 'int'

```

To działanie jest dziedziczone przez metodę `__getattr__` przechwytyującą atrybuty:

```

>>> class C(object):
        def __getattr__(self, name): print(name)

>>> X = C()
>>> X.normal                           # Zwykłe nazwy są wciąż kierowane do
getattr

```

```

normal

>>> X.__add__                                # Podobnie jest w przypadku bezpośrednich
odwołań, ale nie wyrażeń!

__add__

>>> X + 1

TypeError: unsupported operand type(s) for +: 'C' and 'int'

```

Wymogi kodowania obiektów pośredniczących

W bardziej realistycznym scenariuszu delegowania wbudowane operacje, na przykład wyrażenia, nie działają już tak samo jak ich tradycyjne, bezpośrednio wywoływane odpowiedniki. Natomiast bezpośrednie odwołania do nazw wbudowanych metod wciąż działają, jednak równoważne im wyrażenia już nie, ponieważ wywołania przez typy nie działają w przypadku nazw, których nie ma na poziomie klasy i wyżej. Innymi słowy, ta rozbieżność pojawia się tylko w przypadku operacji wbudowanych. Jawne pobrania działają poprawnie:

```

>>> class C(object):
        data = 'mielonka'

        def __getattr__(self, name):
            print('getattr: ' + name)
            return getattr(self.data, name)

>>> X = C()

>>> X.__getitem__(1)                         # Tradycyjne mapowanie działa, ale w nowym
stylu już nie

getattr: __getitem__
'i'

>>> X[1]

TypeError: 'C' object does not support indexing

>>> type(X).__getitem__(X, 1)

AttributeError: type object 'C' has no attribute '__getitem__'

>>> X.__add__('jajka')                      # Tak samo dla operatora +: instancja
pomijana tylko w przypadku wyrażenia

getattr: __add__
'mielonkajajka'

>>> X + 'jajka'

TypeError: unsupported operand type(s) for +: 'C' and 'str'

>>> type(X).__add__(X, 'jajka')

AttributeError: type object 'C' has no attribute '__add__'

```

Ostateczny efekt jest następujący: aby zakodować obiekt pośredniczący, którego interfejs jest po części wykorzystywany przez operacje wbudowane, w klasie w nowym stylu wymagane jest zarówno zdefiniowanie metody `__getattr__` dla zwykłych nazw, jak również *ponowne zdefiniowanie* metod dla wszystkich nazw wykorzystywanych we wbudowanych operacjach —

czy to zakodowanych ręcznie, czy uzyskanych z klas nadrzędnych, czy wygenerowanych przez narzędzia. W przypadku takich definicji wywołania dokonywane zarówno przez instancje, jak i typy są równoważne wbudowanym operacjom. Jednak ponownie zdefiniowane nazwy nie są już kierowane do generycznej metody `__getattr__` obsługującej niezdefiniowane nazwy, nawet w przypadku jawnych wywołań nazw:

```
>>> class C(object):                                     # Nowy styl: wersje 3.x
    i 2.x

        data = 'mielonka'

        def __getattr__(self, name):                      # Przechwytywanie
            zwyklych nazw
                print('getattr: ' + name)
                return getattr(self.data, name)

        def __getitem__(self, i):                          # Ponowne definiowanie
            wbudowanych operacji
                print('getitem: ' + str(i))
                return self.data[i]                         # Uruchomienie wyrazenia
            lub metody setattr

        def __add__(self, other):
            print('dodane: ' + other)
            return getattr(self.data, '__add__')(other)

>>> X = C()
>>> X.upper
getattr: upper
<built-in method upper of str object at 0x0233D670>
>>> X.upper()
getattr: upper
'MIELONKA'
>>> X[1]                                              # Wbudowana operacja
(niejawna)
getitem: 1
'i'
>>> X.__getitem__(1)                                    # Tradycyjny odpowiednik
(jawny)
getitem: 1
'i'
>>> type(X).__getitem__(X, 1)                         # Odpowiednik w nowym
stylu
getitem: 1
'i'
```

```
>>> X + 'jajka'                                # Tak samo dla + i
innych operatorów

dodane: jajka

'mielonkajajka'

>>> X.__add__('jajka')

dodane: jajka

'mielonkajajka'

>>> type(X).__add__(X, 'jajka')

dodane: jajka

'mielonkajajka'
```

Więcej informacji

Do opisanej zmiany powróćmy w rozdziale 40. poświęconym metaklasom, jak również w rozdziale 38. w kontekście zarządzania atrybutami oraz w rozdziale 39. przy opisie dekoratorów. W tym ostatnim przyjrzymy się również sposobom kodowania podstawowych wymaganych metod operatorów w obiektach pośredniczących. Nie jest to niewykonalne zadanie i wystarczy zrealizować je raz a dobrze. Więcej informacji o rodzajach kodów, w których ten problem ma znaczenie, znajduje się w kolejnych rozdziałach, jak również w przykładach pokazanych w rozdziałach 28. i 31.

Ponieważ ten temat rozwiniemy w dalszej części książki, nie będziemy się tutaj więcej nim zajmować. Poniżej znajdują się odnośniki i wskazówki do informacji dotyczących tego problemu (warto też użyć wyszukiwarki):

- *Błąd nr 643841 w Pythonie.* Ten błąd wywołał szeroką dyskusję, a najbardziej oficjalny opis znajduje się pod adresem <http://bugs.python.org/issue643841>. W tym dokumencie został on oznaczony jako problem dla rzeczywistych programów i sklasyfikowany jako przeznaczony do rozwiązania. Jednak zaproponowana poprawka biblioteki lub większa zmiana w Pythonie zostały odrzucone na rzecz prostszej zmiany w dokumentacji opisującej nowe obowiązkowe działanie.
- *Przepisy na narzędzia:* pod adresem <http://code.activestate.com/recipes/252151> znajduje się przepis Active State Python na narzędzie, które automatycznie wypełnia specjalne nazwy metod jako ogólne dyspozitory wywołań w klasie pośredniczącej utworzonej za pomocą przedstawionych w dalszej części tego rozdziału technik wykorzystujących metaklasy. Jednak to narzędzie wciąż pyta użytkownika o podanie nazw metod operatorów, które opakowany obiekt może implementować (musi tak robić, ponieważ komponenty interfejsu opakowanego obiektu mogą być dziedziczone po dowolnych źródłach).
- *Inne źródła:* dzisiaj za pomocą wyszukiwarki można odkrywać różne dodatkowe narzędzia, które w podobny sposób wypełniają klasy pośredniczące przeciążonymi metodami. Jest to powszechny problem. Również w rozdziale 39. zobaczymy, jak bez użycia metaklas, powtarzania kodu i stosowania skomplikowanych technik kodować proste i ogólne klasy nadzędne oferujące wymagane metody lub atrybuty w postaci *domieszek*.

Oczywiście ten temat może ewoluować w miarę upływu czasu, ale jest to problem istniejący od wielu lat. Obecnie typowe klasy obiektów pośredniczących przeciążające dowolne operatory są skutecznie zakłócane przez klasy w nowym stylu. Takie klasy, zarówno w wersji 2.x, jak i 3.x, wymagają kodowania lub generowania obiektów metod opakowujących wszystkie niejawnie wywoływane metody operatorów, które mają być obsługiwane. Nie jest to idealne rozwiązanie, ponieważ niektóre klasy pośredniczące mogą wymagać dziesiątek metod opakowujących (potencjalnie ponad 50!), ale odzwierciedla cele zaprojektowania klas w nowym stylu.



Rozdział 40. poświęcony *metaklasom* zawiera dodatkowy opis tego problemu i jego podstaw. Jak się przekonamy, takie działanie wbudowanych operacji można zakwalifikować jako specjalny przypadek w *dziedziczeniu* klas w nowym stylu. Jego dokładne poznanie wymaga większej wiedzy o metaklasach niż zawarta w tym rozdziale. Jest to efekt uboczny wprowadzenia metaklas — stały się bardziej niezbędne do szerszych zastosowań, niż przewidywali ich twórcy.

Zmiany w modelu typów

Zajmijmy się kolejną zmianą w klasach w nowym stylu. W zależności od oceny w klasach w nowym stylu różnica między typami a klasami została zupełnie zatarta.

Klasy są typami

Obiekt `type` generuje klasy będące jego instancjami, natomiast klasy generują instancje swoich typów. Oba pojęcia są typami, ponieważ generują instancje. W rzeczywistości nie ma żadnej różnicy między typami wbudowanymi, jak listy i ciągi znaków, a klasami zdefiniowanymi przez użytkownika. Dlatego można tworzyć klasy pochodne od wbudowanych typów, jak pokazano wcześniej w tym rozdziale. Klasę pochodną od wbudowanego typu, na przykład `list`, kwalifikuje się jako klasę w nowym stylu i jest to nowy typ zdefiniowany przez użytkownika.

Typy są klasami

Nowe typy generujące klasy można w Pythonie kodować jako metaklasy, z którymi zapoznamy się w dalszej części rozdziału. Są to klasy pochodne od typów zdefiniowanych przez użytkownika, kodowane za pomocą zwykłej instrukcji `class` i kontrolujące tworzenie klas, które są ichinstancjami. Jak się przekonamy, metaklasy są zarówno klasami, jak i typami, jednak wyróżniają się na tyle, że uzasadniają przeobrażenie poprzedniego podziału typ/klasa w metaklasa/klasa, choć kosztem dodatkowej złożoności w normalnych klasach.

Oprócz możliwości tworzenia klas potomnych typów wbudowanych i kodowania metaklas jednym z najbardziej praktycznych kontekstów, w których konsolidacja typu i klasy staje się oczywista, jest jawne sprawdzanie typów. W klasycznych klasach Pythona 2.x typem instancji klasy jest generyczna instancja, natomiast typy obiektów wbudowanych są już bardziej specjalizowane.

```
C:\code> c:\python27\python
>>> class C: pass                                # Klasyczne klasy w wersji 2.x
>>> I = C()                                         # Instancje są tworzone z klas
>>> type(I), I.__class__
(<type 'instance'>, <class __main__.C at 0x02399768>)
>>> type(C)                                         # Jednak klasy nie są tym
samym co typy
<type 'classobj'>
>>> C.__class__
AttributeError: class C has no attribute '__class__'
>>> type([1, 2, 3]), [1, 2, 3].__class__
(<type 'list'>, <type 'list'>)
>>> type(list), list.__class__
```

```
(<type 'type'>, <type 'type'>)
```

Jednak w przypadku klas w nowym stylu w 2.x typem instancji klasy jest jej klasa, ponieważ klasy są w rzeczywistości typami zdefiniowanymi przez użytkownika — typem instancji jest jej klasa, a typ klasy zdefiniowanej przez użytkownika jest taki sam jak typ wbudowanego obiektu type. Klasy posiadają również atrybut `__class__`, ponieważ same są instancjami klasy type.

```
C:\code> c:\python27\python
>>> class C(object): pass
... # Klasy w nowym stylu w 2.x
>>> I = C()
>>> type(I), I.__class__
... # Typem instancji jest jej klasa
(<class '__main__.C'>, <class '__main__.C'>)
>>> type(C), C.__class__
... # Klasy są typami zdefiniowanymi przez użytkownika
(<type 'type'>, <type 'type'>)
```

Ta sama zasada obowiązuje w przypadku wszystkich klas w Pythonie 3.x, ponieważ tutaj wszystkie klasy są klasami w nowym stylu, nawet jeśli nie posiadają żadnych jawnie zadeklarowanych klas nadzędnych. W rzeczywistości rozróżnienie między klasami zdefiniowanymi przez użytkownika a typami wbudowanymi w 3.x zupełnie przestaje istnieć.

```
C:\code> c:\python33\python
>>> class C: pass
...
>>> I = C() # W 3.x wszystkie klasy są w nowym stylu
>>> type(I), I.__class__ # Typem instancji jest jej klasa
(<class '__main__.C'>, <class '__main__.C'>)
>>> type(C), C.__class__ # Klasa jest typem, a typ jest klasą
(<class 'type'>, <class 'type'>)
>>> type([1, 2, 3]), [1, 2, 3].__class__ # Klasy i typy wbudowane działają tak samo
(<class 'list'>, <class 'list'>)
>>> type(list), list.__class__ # Klasy i typy wbudowane działają tak samo
(<class 'type'>, <class 'type'>)
```

Jak widać, w 3.x klasy są typami, ale typy są również klasami. Z technicznego punktu widzenia każda klasa jest generowana przez *metaklasę*. Najczęściej jest to klasa `type` albo jej klasa potomna z metodami przeciążonymi w celu modyfikacji zachowania tworzonych klas. Ta zmiana może spowodować problemy z kompatybilnością kodu wykorzystującego kontrolę typów, ale jest istotna dla twórców narzędzi dla programistów. Więcej informacji na temat metaklas przedstawię w dalszej części tego rozdziału oraz omówię je bardziej szczegółowo w rozdziale 40.

Konsekwencje z perspektywy kontroli typów

Oprócz możliwości modyfikowania typów wbudowanych oraz mechanizmu metaklas połączenie klas z typami w klasach w nowym stylu może nieść zagrożenia w postaci efektów ubocznych w kodzie wykorzystującym jawną kontrolę typów. Na przykład w Pythonie 3.x typy instancji klas zdefiniowanych przez użytkownika można porównywać bezpośrednio i porównania te dadzą użyteczne informacje, tak samo jak w przypadku obiektów wbudowanych. Właściwość ta bierze się z faktu, że klasy są obecnie typami, a typ instancji jest jej klasą.

```
C:\code> c:\python33\python
>>> class C: pass
>>> class D: pass
...
>>> c, d = C(), D()
>>> type(c) == type(d)                                # 3.x: porównanie klas instancji
False
>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)
>>> c1, c2 = C(), C()
>>> type(c1) == type(c2)
True
```

W przypadku klasycznych klas Pythona 2.x porównywanie typu instancji jest praktycznie bezużyteczne, ponieważ wszystkie instancje mają ten sam typ „instancji”. Aby efektywnie porównać typy, należy zastosować wartość atrybutu `__class__` instancji (w celu zachowania kompatybilności w 3.x ta technika również zadziała, ale w tym przypadku jej stosowanie nie jest już niezbędne).

```
C:\misc> c:\python27\python
>>> class C: pass
>>> class D: pass
...
>>> c, d = C(), D()
>>> type(c) == type(d)                                # 2.x: wszystkie klasyczne instancje
są tego samego typu
True
>>> c.__class__ == d.__class__                         # Należy jawnie porównywać klasy
False
>>> type(c), type(d)
(<type 'instance'>, <type 'instance'>)
```

```

>>> c.__class__, d.__class__
(<class '__main__.C' at 0x024585A0>, <class '__main__.D' at 0x024588D0>)

Jak można się spodziewać, klasy w nowym stylu w 2.x działają dokładnie tak samo jak w 3.x — porównanie typów instancji automatycznie powoduje porównanie ich klas.

C:\misc> c:\python27\python
>>> class C(object): pass
>>> class D(object): pass
...
>>> c, d = C(), D()
>>> type(c) == type(d)           # 2.x w nowym stylu: działają tak samo
jak w 3.x
False
>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)

```

Jednak należy pamiętać, o czym wspominałem w tej książce wielokrotnie, że bezpośrednie porównywanie typów instancji jest z reguły niewłaściwym podejściem w programach w Pythonie — powinniśmy wykorzystywać interfejsy realizowane przez obiekty, nie ich typy. W rzadkich sytuacjach, gdy jesteśmy zmuszeni do jawnej kontroli typów, powinniśmy raczej stosować funkcję wbudowaną `isinstance`. Jednak warto znać model typów zaimplementowany w Pythonie, ponieważ to pozwoli lepiej zrozumieć model klas.

Wszystkie obiekty dziedziczą po klasie object

Jedną z najważniejszych cech modelu klas w nowym stylu jest to, że wszystkie klasy bezpośrednio lub pośrednio dziedziczą po klasie `object`, a z faktu, że typy są również klasami, wynika, że każdy obiekt bezpośrednio lub pośrednio dziedziczy po klasie wbudowanej `object`. Weźmy na przykład następujący kod w Pythonie 3.x:

```

>>> class C: pass                  # Klasa w nowym stylu
>>> X = C()
>>> type(X), type(C)             # Typ jest klasą instancji
(<class '__main__.C'>, <class 'type'>)

```

Podobnie jak poprzednio, typem *instancji* klasy jest jej klasa, a typem *klasy* jest klasa `type`, ponieważ typy i klasy zostały scalone. Należy również pamiętać, że zarówno instancja, jak i klasa dziedziczą po klasie `object`, będącej jawną lub niejawną klasą nadzczną dla wszystkich klas:

```

>>> isinstance(X, object)
True
>>> isinstance(C, object)          # Klasa zawsze dziedziczą po object
True

```

W wersji 2.x powyższa instrukcja zwraca ten sam wynik zarówno w przypadku klasy zwykłej, jak i w nowym stylu. Jednak typ wyniku jest inny. Co więcej, jak się później przekonamy, klasa `object` nie jest dodawana ani nie jest obecna w krotce `__bases__` zwykłej klasy w wersji 2.x, zatem nie jest to prawdziwa klasa nadzędna.

Ta sama zasada obowiązuje w stosunku do klas wbudowanych, jak listy czy ciągi znaków, ponieważ w modelu klas w nowym stylu typy są klasami, a instancje typów wbudowanych również dziedziczą po klasie wbudowanej `object`.

```
>>> type('spam'), type(str)
(<class 'str'>, <class 'type'>
>>> isinstance('spam', object)           # To samo dotyczy klas wbudowanych
True
>>> isinstance(str, object)
True
```

W rzeczywistości klasa `type` również dziedziczy po klasie `object`, a klasa `object` dziedziczy po klasie `type` i choć obydwie są niezależnymi obiektami, to ta cykliczna zależność pozwala ująć w jedną całość tę najważniejszą cechę klas w nowym stylu: typy są klasami i służą do generowania klas.

```
>>> type(type)                         # Wszystkie klasy są typami, a typy są
      klasami
<class 'type'>
>>> type(object)
<class 'type'>
>>> isinstance(type, object)           # Wszystkie klasy dziedziczą po object,
      nawet klasa type
True
>>> isinstance(object, type)          # Typy tworzą klasy, a type jest klasą
True
>>> type is object
False
```

Implikacje wynikające z użycia metod domyślnych

W powyższym przykładzie nie widać tego wyraźnie, ale z opisywanego modelu wynika kilka praktycznych implikacji. Przede wszystkim trzeba pamiętać o metodach domyślnych zawartych w jawniej lub niejawnie klasie głównej `object` wyłącznie w klasie w nowym stylu:

```
c:\code> py -2
>>> dir(object)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__',
 '__hash__',
 '__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
```

```

>>> class C: pass
>>> C.__bases__                                # Zwykłe klasy nie pochodzą od
klasy object
()
>>> X = C()
>>> X.__repr__
AttributeError: C instance has no attribute '__repr__'

>>> class C(object): pass                      # Klasy w nowym stylu dziedziczą
domyślne cechy klasy object

>>> C.__bases__
(<type 'object'>,)
>>> X = C()
>>> X.__repr__
<method-wrapper '__repr__' of C object at 0x00000000020B5978>
c:\code> py -3
>>> class C: pass                            # To oznacza, że w wersji 3.x
wszystkie klasy otrzymują domyślne cechy
>>> C.__bases__
(<class 'object'>,)
>>> C().__repr__
<method-wrapper '__repr__' of C object at 0x0000000002955630>

```

W praktyce ten model powoduje, że mamy do czynienia z mniejszą liczbą przypadków szczególnych, niż ma to miejsce w klasycznym modelu klas, a ponadto pozwala na pisanie kodu wykorzystującego klasę nadzczną `object` (zakładając, że jest to „kotwica” w niektórych opisanych dalej wbudowanych rolach funkcji `super`, oraz przekazując jej domyślne metody do wykonywania domyślnych operacji). Szczegółowe omówienie możliwości wynikających z tego faktu znajduje się w dalszej części książki, tymczasem przejdźmy do innych zmian wprowadzonych w klasach w nowym stylu.

Zmiany w dziedziczeniu diamentowym

Jedną z widocznych zmian w klasach w nowym stylu jest nieco odmienny algorytm wyszukiwania nazw w tak zwanym wielodziedziczeniu *diamentowym*, to znaczy takim, w którym klasa dziedziczy po kilku klasach, mających wspólnego przodka. (Nazwa pochodzi od diamentu, jaki pojawia się na rysunku drzewa dziedziczenia).

Dziedziczenie diamentowe jest zaawansowanym wzorcem projektowym, rzadko stosowanym w Pythonie w sposób bezpośredni i do tej pory przez nas nieomawianym, zatem nadszedł czas, aby nieco zgłębić ten temat.

Klasyczne klasy (domyślne w wersji 2.x): wyszukiwanie DFLR

Procedura wyszukiwania nazw działa w pierwszej kolejności w pionie, a następnie od lewej do prawej. Innymi słowy, drzewo dziedziczenia jest najpierw przeszukiwane w głąb, z zachowaniem trzymania się lewej strony, a dopiero jeśli nazwa nie zostanie odszukana,

procedura wraca na dół drzewa i klasy są przeszukiwane od lewej do prawej. Ta kolejność wyszukiwania jest określana mianem DFLR, będącym skrótem od *deep first, then left right* (najpierw w głąb, potem od lewej na prawą).

Klasy w nowym stylu (opcjonalne w wersji 2.x, automatyczne w 3.x): wyszukiwanie MRO

Ta procedura przeszukiwania przebiega głównie w poprzek struktury diamentowej — Python najpierw poszukuje wszerz w klasach nadrzędnych pierwszego poziomu, od lewej do prawej, a następnie przechodzi piętro wyżej. Innymi słowy, najpierw przeszukiwane są poziomy, dopiero potem następuje przejście wyżej. Ta kolejność wyszukiwania jest określana mianem algorytmu MRO w nowym stylu (skrót od *method resolution order*; często stosowany przy porównywaniu z wyszukiwaniem DFLR). Mimo że nazwa nawiązuje do metod, algorytm ten dotyczy również atrybutów.

Algorytm wyszukiwania nazw jest nieco bardziej skomplikowany — bardziej formalnie rozwiniemy go w dalszej części książki — ale tyle szczegółów powinno wystarczyć programistom. Jak i poprzednio algorytm ten daje wiele korzyści w klasach w nowym stylu, ale może zakłócić działanie klas w istniejącym kodzie.

Dzięki tej zmianie klasy na niższym poziomie dziedziczenia mogą przesłaniać nazwy zdefiniowane w klasach nadrzędnych, niezależnie od tego, na której pozycji zostały zadeklarowane na liście klas nadrzędnych. Co więcej, reguła wielokrotnego dziedziczenia pozwala uniknąć wielokrotnego przeszukiwania tej samej klasy nadrzędnej, jeśli znajduje się ona na liście dziedziczenia większej liczby klas w drzewie. Jest to zdecydowanie lepszy algorytm niż DFLR, ale dotyczy tylko niewielkiego fragmentu kodu użytkownika. Jak się przekonamy, sam model klas w nowym stylu sprawia, że dziedziczenie diamentowe jest częściej spotykane, a algorytm MRO jest ważniejszy.

Jednocześnie nowy algorytm MRO w inny sposób lokalizuje atrybuty, co potencjalnie skutkuje niekompatybilnością zwykłych klas w wersji 2.x. Sprawdźmy za pomocą przykładowego kodu, jak te różnice wyglądają w praktyce.

Implikacje wynikające z dziedziczenia diamentowego

Aby pokazać, czym się różni algorytm MRO w klasach w nowym stylu, przeanalizujmy prosty pseudokod dziedziczenia diamentowego zwykłych klas. Klasa D dziedziczy po klasach B i C, które mają wspólnego przodka — klasę A.

```
>>> class A:           attr = 1      # Klasyczne klasy (Python 2.x)
>>> class B(A):        pass          # B i C dziedziczą po A
>>> class C(A):        attr = 2
>>> class D(B, C):     pass          # Sprawdza w A, a następnie w C
>>> x = D()
>>> x.attr                         # Kolejność przeszukiwania: x, D, B,
A
1
```

W tym przypadku poszukiwana nazwa `x.attr` została znaleziona w klasie A, ponieważ w klasach typu klasycznego algorytm DFLR przeszukuje drzewo wzwyż, a dopiero potem w prawo, przez co Python przeszuka klasy w kolejności x, D, B, A, C i ponownie A. W tym przypadku przeszukiwanie kończy się po odszukaniu `attr` w klasie A (powyżej B), przez co nigdy nie dociera do C.

W klasach w nowym stylu dziedziczących po klasie `object`, czyli we wszystkich klasach w 3.x, algorytm wyszukiwania nazw jest inny: Python najpierw przeszuka klasę C (na prawo od B), a

dopiero potem A (powyżej B). Algorytm MRO przeszukuje więc klasy w kolejności x, D, B, C i na koniec A. W tym przypadku poszukiwanie atrybutu `attr` kończy się w klasie C.

```
>>> class A(object):      attr = 1      # Nowy styl (dziedziczenie po
   "object" niewymagane w 3.x)

>>> class B(A):          pass

>>> class C(A):          attr = 2

>>> class D(B, C):        pass          # Sprawdza w C, a następnie w A

>>> x = D()

>>> x.attr                  # Kolejność przeszukiwania: x, D, B,
C

2
```

Ta zmiana procedury wyszukiwania nazw w drzewie dziedziczenia jest oparta na założeniu, że jeśli wmiieszamy nazwy zdefiniowane w klasę C na niższym poziomie drzewa, chcielibyśmy, aby miały wyższy priorytet niż nazwy zdefiniowane w klasie A. Przyjmuje się, że jeśli ktoś zdefiniował klasę C z nazwami przesłaniającymi nazwy klasy A, zrobił to po to, by używać nazw z klasy C. Tak się dzieje zawsze w sytuacji pojedynczego dziedziczenia, ale już przy wielodziedziczeniu diamentowym w przypadku klasycznego modelu klas tak nie musi być — można nawet nie wiedzieć, że klasa C jest wymieszana w opisany wyżej sposób.

W praktyce tak skonstruowane dziedziczenie, w którym nazwy klasy C przesłaniają nazwy klasy A, po której dziedziczą, oznacza intencję programisty, aby były użyte nazwy klasy C. W przeciwnym razie użycie klasy C jest właściwie bezzasadne w kontekście dziedziczenia diamentowego — nie ma wpływu na nazwy zdefiniowane w klasie A, zatem z klasy C zostaną użyte tylko te nazwy, które są dla niej unikalne.

Jawne rozwiązywanie konfliktów

Oczywiście problemem założeń jest to, że zakładają one różne rzeczy. Jeśli taka modyfikacja kolejności wyszukiwania wydaje nam się zbyt subtelna, by ją zapamiętać, lub jeśli chcemy mieć większą kontrolę nad procesem wyszukiwania, możemy zawsze wymusić wybór atrybutu z dowolnego miejsca drzewa, przypisując lub w inny sposób wymieniając ten, który chcemy, w miejscu, w którym klasy są mieszane. Na przykład w poniższym kodzie wybierany jest nowy styl wyszukiwania poprzez jawne odwzorowywanie nazw:

```
>>> class A:                  attr = 1      # Model klasyczny

>>> class B(A):              pass

>>> class C(A):              attr = 2

>>> class D(B, C):           attr = C.attr    # Wybór C, na prawo

>>> x = D()

>>> x.attr                  # Działa jak klasy w nowy stylu
(wszystkie klasy w 3.x)

2
```

W powyższym kodzie drzewo klas z modelu klasycznego emuluje kolejność wyszukiwania klas w nowym stylu. Przypisanie do atrybutu w klasie D wybiera jego wersję z klasy C, odwracając tym samym normalną ścieżkę wyszukiwania dziedziczenia (`D.attr` będzie się znajdować najbliżej w drzewie). Klasy w nowym stylu mogą w podobny sposób emulować model klasyczny, wybierając atrybut z góry w miejscu, w którym klasy mieszają się ze sobą.

```
>>> class A(object):      attr = 1      # Nowy styl
```

```

>>> class B(A):
                    pass
>>> class C(A):
                    attr = 2
>>> class D(B,C):
                    attr = B.attr    # Wybór A.attr, z góry
>>> x = D( )
>>> x.attr
(domyślny w wersji 2.x)
1

```

Jeśli mamy ochotę zawsze rozwiązywać konflikty w ten sposób, możemy w dużym stopniu zignorować różnice w kolejności wyszukiwania i nie polegać na założeniach dotyczących tego, co mieliśmy na myśli, tworząc kod klas.

Oczywiście atrybuty pobrane w ten sposób mogą również być funkcjami metod — metody są normalnymi obiektami, które można przypisywać.

```

>>> class A:
...     def meth(s): print ('A.meth')
>>> class C(A):
...     def meth(s): print ('C.meth')
>>> class B(A):
...     pass
>>> class D(B,C): pass
kolejności wyszukiwania                                # Wykorzystanie domyślnej
>>> x = D( )                                         # Różni się dla typów klas
>>> x.meth( )                                       # Domyślnie kolejność klasyczna
w wersji 2.x
A.meth
>>> class D(B,C): meth = C.meth                      # Wybór metody klasy C: nowy
styl (i wersja 3.x)
>>> x = D( )
>>> x.meth( )
C.meth
>>> class D(B,C): meth = B.meth
klasyczny                                              # Wybór metody klasy B: model
>>> x = D( )
>>> x.meth( )
A.meth

```

W powyższym kodzie wybieramy metody, w jawnym sposobie przypisując wartości do zmiennych znajdujących się niżej w drzewie. Możemy również wywołać pożądaną klasę w sposób jawnym. W praktyce ten wzorzec może zyskać na popularności, w szczególności dla elementów takich, jak konstruktory.

```
class D(B,C):
```

```

def meth(self):                                # Redefiniowanie niżej
    ...
    C.meth(self)                            # Wybór metody klasy C przez
wywołanie

```

Taki wybór przez przypisanie lub wywołanie w miejscu mieszania może w rezultacie odizolować nasz kod od różnic w typach klas. Dotyczy to oczywiście wyłącznie atrybutów obsługiwanych w ten sposób, jednak jawne rozwiązywanie konfliktów zapewnia, że nasz kod nie będzie się zachowywał inaczej w różnych przyszłych wersjach Pythona, przynajmniej pod względem rozwiązywania konfliktów atrybutów. Innymi słowy, jest to technika zapewniająca *przenośność* klas, które muszą działać zarówno w modelu w nowym stylu, jak i klasycznym.



Jawność jest lepsza niż niejawność — również w modelu rozwiązywania konfliktów. Nawet bez rozbieżności między modelem klasycznym a nowym stylem, technika ta może się czasem przydać w scenariuszach dziedziczenia wielokrotnego. Jeśli chcemy uzyskać część klasy nadzędnej z lewej strony oraz część klasy nadzędnej z prawej strony, musimy przekazać Pythonowi, który z atrybutów o tych samych nazwach ma wybrać, wykorzystując do tego jawne przypisania w klasach podrzędnych. Powrócimy do tego zagadnienia w podrozdziale poświęconym pułapkom związanym z klasami pod koniec tego rozdziału.

Warto również zauważyć, iż wzorce wielokrotnego dziedziczenia po jednej klasie nadzędnej mogą w niektórych przypadkach być bardziej problematyczne, niż pokazano to tutaj (co na przykład stanie się, jeśli zarówno B, jak i C mają wymagane konstruktory wywołujące A?). Tego typu sytuacje są rzadkością w Pythonie, zatem odłożymy je na koniec rozdziału. Oprócz udostępniania klas nadzędnych w drzewach z pojedynczym dziedziczeniem super oferuje tryb rozstrzygania konfliktów nazw w drzewach z wielokrotnym dziedziczeniem poprzez wywoływanie metod w kolejności zgodnej z MRO — przyjmując założenie, że taka kolejność ma sens również w tym kontekście.

Zakres zmian kolejności wyszukiwania

Podsumowując, wzorzec dziedziczenia diamentowego jest przeszukiwany w inny sposób w klasach typu klasycznego i w klasach w nowym stylu i ta zmiana nie zachowuje zgodności wstępco. Należy jednak pamiętać, że ma ona wpływ na wielodziedziczenie typu diamentowego i klasy w nowym stylu w większości przypadków będą działały dokładnie tak samo jak klasy typu klasycznego. Co więcej, niewykluczone jest, że problem różnicy algorytmu wyszukiwania ma znaczenie bardziej teoretyczne. W końcu aż do Pythona 2.2 nikt nie zainteresował się błędną koncepcją zastosowaną w algorytmie wyszukiwania, a na wprowadzenie go w standardzie czekaliśmy aż do Pythona 3.0, co sugeruje, że niewiele jest kodu Pythona, w którym ten problem może wystąpić.

Niemniej jednak należy pamiętać, że choć niewielu programistów w sposób jawnym i świadomym korzysta z diamentowego wzorca dziedziczenia, to *wszystkie* przypadki wielodziedziczenia w Pythonie 3.x są implementacją właśnie tego wzorca. Dzieje się tak z tego powodu, że wszystkie klasy dziedziczą po klasie `object`, która stanowi odpowiednik klasy A z naszego powyższego przykładu. Z tego powodu nowy algorytm MRO nie tylko wprowadza modyfikację semantyki działania dziedziczenia, lecz również *optymalizację wydajności* — dzięki uniknięciu wielokrotnego wyszukiwania nazwy w tej samej klasie występującej w drzewie wielokrotnie.

Co nie mniej istotne, domyślnie wykorzystana klasa nadzędna `object` dostarcza wszystkim obiektom wielu *metod* wykorzystywanych przez operacje wbudowane, jak `__str__` czy `__repr__`. Aby zapoznać się z listą metod oferowanych przez klasę `object`, wystarczy wywołać `dir(object)`. Gdyby nie wprowadzono algorytmu MRO, w klasach w nowym stylu metody te nie mogłyby być efektywnie przesłonięte, chyba że klasy je redefiniujące zostałyby zadeklarowane

na pierwszym miejscu na liście klas nadzędnych. Innymi słowy, zmiana algorytmu wyszukiwania nazw w dziedziczeniu diamentowym to wręcz warunek konieczny efektywnego funkcjonowania klas w nowym stylu.

Bardziej praktyczny przykład wykorzystania metod klasy `object` w dziedziczeniu w Pythonie 3.x można znaleźć w definicji klasy `ListTree` z modułu `lister.py` w poprzednim rozdziale, w przykładzie `classtree.py` w rozdziale 29. oraz w następnym podrozdziale.

Więcej o kolejności odwzorowywania nazw

Aby prześledzić domyślne dziedziczenie w klasach w nowym stylu, można również wykorzystać nowy atrybut `class.__mro__` wspomniany w prezentowanych w poprzednim rozdziale przykładach wyświetlania informacji o klasach. Z technicznego punktu widzenia jest to przydatne rozszerzenie, które teraz dokładniej zbadamy. Atrybut ten zawiera informacje o kolejności przeszukiwania drzewa hierarchii klas w nowym stylu. Algorytm MRO opiera się na algorytmie linearyzacji klasy nadzędnej C3, opracowanym w języku Dylan, a następnie zaadaptowanym w innych językach, m.in. Python 2.3 i Perl 6.

Algorytm MRO

W tej książce specjalnie został pominięty pełny opis algorytmu MRO, ponieważ dla większości programistów nie ma on znaczenia (ma jedynie wpływ na dziedziczenie diamentowe, rzadko stosowane w rzeczywistych programach), wygląda inaczej w wersjach 2.x i 3.x, a szczegóły jego działania są zbyt tajemnicze i akademickie. Z reguły w tej książce unikam formalnych opisów algorytmów i preferuję ich nieformalne objaśnianie za pomocą przykładów.

Z drugiej strony część czytelników może interesować formalna teoria algorytmu MRO w klasach w nowym stylu. Jego szczegółowy opis można znaleźć w internecie. Wystarczy przeszukać podręczniki do Pythona i internet pod kątem frazy „MRO”. W skrócie algorytm ten wygląda następująco:

1. Utwórz listę wszystkich klas nadzędnych dla danej instancji, wykorzystując klasyczny algorytm DFLR. Jeżeli klasa występuje więcej niż jeden raz, dołącz ją wielokrotnie do listy.
2. Przeszukaj listę pod kątem duplikatów klas i usuń wszystkie z wyjątkiem ostatniej.

Wynikowa lista MRO zawiera daną klasę, jej klasy nadzędne, wszystkie klasy nadzędne w hierarchii aż do klasy głównej znajdującej się na szczycie drzewa. Klasy są ułożone w takiej kolejności, że każda z nich pojawia się przed swoimi klasami nadzędnymi, a klasy nadzędne są ułożone w takiej kolejności, w jakiej pojawiają się w krotce `__bases__`.

Ponieważ wspólne klasy nadzędne w dziedziczeniu *diamentowym* pojawiają się tylko na ostatnio odwiedzonych pozycjach, więc gdy lista MRO jest później wykorzystywana do dziedziczenia atrybutów, niższe klasy są przeszukiwane w pierwszej kolejności. Co więcej, każda klasa jest dodawana, a przez to odwiedzana tylko raz, bez względu na to, ile innych klas do niej prowadzi.

W dalszej części rozdziału poznamy zastosowania tego algorytmu, z wykorzystaniem funkcji `super` wyłącznie. Jest to wbudowana funkcja powodująca, że opis algorytmu MRO jest lekturą obowiązkową dla tych, którzy zdecydują się jej używać i chcą się dokładnie dowiedzieć, jak wywołuje ona metody. Jak się przekonamy, pomimo swojej nazwy funkcja ta wywołuje zgodnie z algorytmem MRO następną klasę, która wcale nie musi być klasą nadzędzią.

Śledzenie algorytmu MRO

Aby jedynie ogólnie dowiedzieć się, jak w dziedziczeniu klas w nowym stylu porządkowane są klasy nadzędne, wystarczy sprawdzić atrybut `class.__mro__` klasy w nowym stylu (a więc

każdej klasy w wersji 3.x). Jest to krotka, którą Python wykorzystuje do wyszukiwania atrybutów w klasach nadzędnych. W rzeczywistości atrybut ten zawiera kolejność dziedziczenia klas w nowym stylu i zazwyczaj szczegóły działania MRO, których potrzebuje większość użytkowników Pythona.

Poniżej przedstawionych jest kilka reprezentatywnych przykładów dla wersji 3.x. W przypadku dziedziczenia *diametrowego* przeszukiwanie odbywa się w nowej kolejności, którą analizowaliśmy — najpierw wszerz drzewa, potem w górę, zgodnie z algorytmem MRO dla klas w nowym stylu, dostępnych zawsze w wersji 3.x i opcjonalnych w 2.x.

```
>>> class A: pass
>>> class B(A): pass          # Dziedziczenie diamentowe: kolejność jest
                               inna dla klas w nowym stylu
>>> class C(A): pass          # Najpierw wszerz na niższych poziomach
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>)
```

Jednak w dziedziczeniu *innym niż diamentowe* kolejność przeszukiwania jest taka, jak była od zawsze (ale obejmuje dodatkową klasę główną `object`) — najpierw w górę, potem w prawo (ang. *depth first and left to right*, w skrócie DFLR; ten model jest stosowany dla wszystkich zwykłych klas w wersji 2.x):

```
>>> class A: pass
>>> class B(A): pass          # Dziedziczenie inne niż diamentowe: kolejność
                               taka sama jak dla zwykłych klas
>>> class C: pass             # Najpierw w głąb, potem od lewej na prawą.
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>,
 <class '__main__.C'>, <class 'object'>)
```

Na przykład kolejność MRO w poniższym drzewie jest taka sama jak wcześniej w dziedziczeniu diamentowym zgodnie z algorytmem DFLR:

```
>>> class A: pass
>>> class B: pass              # Inne dziedziczenie niediamamentowe: DFLR
>>> class C(A): pass
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>)
```

Warto zwrócić uwagę, że na końcu MRO pojawia się niejawną klasą nadzijną `object`. Jak widzieliśmy, jest ona automatycznie dodawana nad najwyższymi klasami w nowym stylu w wersji 3.x (i opcjonalnie w wersji 2.x):

```

>>> A.__bases__          # Odnośniki do klasy nadzędnej: object
dwukrotnie jako klasa główna

(<class 'object'>,)

>>> B.__bases__
(<class 'object'>,)

>>> C.__bases__
(<class '__main__.A'>,)

>>> D.__bases__
(<class '__main__.B'>, <class '__main__.C'>)

```

Z technicznego punktu widzenia niejawną klasą nadzędną zawsze tworzy diamentową strukturę w dziedziczeniu wielokrotnym, nawet jeśli nie robią tego klasy. Klasy są przeszukiwane tak samo jak wcześniej, jednak algorytm MRO w nowym stylu gwarantuje, że klasa `object` jest odwiedzana na końcu, dzięki czemu klasy użytkownika mogą nadpisywać domyślne ustawienia:

```

>>> class X: pass
>>> class Y: pass
>>> class A(X): pass          # Dziedziczenie niediametowe: najpierw w
głąb, potem od lewej na prawą
>>> class B(Y): pass          # Niejawną klasą object zawsze tworzy
diament
>>> class D(A, B): pass
>>> D.mro()
[<class '__main__.D'>, <class '__main__.A'>, <class '__main__.X'>,
<class '__main__.B'>, <class '__main__.Y'>, <class 'object'>]
>>> X.__bases__, Y.__bases__
((<class 'object'>,), (<class 'object'>,))
>>> A.__bases__, B.__bases__
((<class '__main__.X'>,), (<class '__main__.Y'>,))

```

Atrybut `class.__mro__` jest dostępny tylko w klasach w nowym stylu. Nie ma go w klasach w wersji 2.x, chyba że dziedziczą one klasę `object`. Mówiąc ściślej, klasy w nowym stylu mają również metodę `class.mro` zastosowaną dla odmiany w poprzednim przykładzie. Metoda ta jest wywoływana w chwili tworzenia instancji, a zwracany przez nią wynik jest listą wykorzystywaną do inicjowania atrybutu `__mro__` podczas tworzenia klasy (metodę tę można dostosowywać za pomocą metaklas opisanych w dalszej części książki). Można również wybierać nazwy MRO, jeżeli wyświetlane dane obiektów klas są zbyt szczegółowe. Jednak w tej książce obiekty są zazwyczaj pokazywane dla przypomnienia ich rzeczywistej formy:

```

>>> D.mro() == list(D.__mro__)
True
>>> [cls.__name__ for cls in D.__mro__]
['D', 'A', 'X', 'B', 'Y', 'object']

```

Niezależnie od tego, jak odwołujemy się do klasy lub ją wyświetlamy, ścieżki MRO można wykorzystywać w procesie rozwiązywania niejasności oraz w narzędziach, które muszą imitować kolejność przeszukiwania drzewa dziedziczenia w Pythonie. W następnym podrozdziale opisana jest druga rola.

Przykład – wiązanie atrybutów ze źródłami dziedziczenia

Na końcu poprzedniego rozdziału stwierdziliśmy, że programy przeglądające drzewo klas, na przykład przedstawiony tam kod wyświetlający drzewo, mogą skorzystać na algorytmie MRO. Po zakodowaniu program podał fizyczne lokalizacje atrybutów w drzewie klas. Jednak wiążąc listę odziedziczonych atrybutów zawartych w wyniku metody `dir` z liniową sekwencją MRO (lub DFLR przypadku zwykłych klas), tego rodzaju narzędzia mogą bezpośrednio kojarzyć atrybuty z klasami, po których są dziedziczone. To też jest funkcjonalność przydatna dla programistów.

Nie będziemy tu zmieniać kodu wyświetlającego drzewo. W pierwszym ważnym kroku wykorzystamy przedstawiony niżej plik `mapattrs.py` implementujący narzędzia, które można wykorzystywać do wiązania atrybutów z ich źródłami dziedziczenia. Dodatkowo funkcja `mapattrs` pokazuje, jak mechanizm dziedziczenia w rzeczywistości przeszukuje atrybuty w obiektach drzewa klas, choć algorytm MRO w nowym stylu w dużej mierze robi to automatycznie za nas:

```
"""
Plik mapattrs.py (3.x + 2.x)

Główne narzędzie: funkcja mapattrs wiąże wszystkie atrybuty
instancji: własne i odziedziczone z instancją lub z klasą, z której pochodzą.

Przyjęto założenie, że funkcja dir zwraca wszystkie atrybuty instancji.

Aby zasymulować dziedziczenie wykorzystuje krotkę MRO klasy,
zawierającą kolejność przeszukiwania klas w nowym stylu (wszystkich
w wersji 3.x) lub rekurencyjnie przemierza drzewo w celu
odgadnięcia kolejności DFLR przeszukiwania zwykłych klas w wersji 2.x.

Uwaga: funkcja inheritance daje niezależne od wersji uporządkowanie klas.

Dostępne są różne narzędzia słownikowe wykorzystujące wyrażenia w wersjach
3.x/2.x.

"""

import pprint

def trace(X, label='', end='\n'):
    print(label + pprint.pformat(X) + end)           # Wyświetlenie w eleganckim
    formacie

def filterdictvals(D, V):
    """
    Słownik D z usuniętymi wpisami dla wartości V.
    filterdictvals(dict(a=1, b=2, c=1), 1) => {'b': 2}
    """

    pass
```

```

"""
    return {K: V2 for (K, V2) in D.items() if V2 != V}
def invertdict(D):
    """
        Słownik D z wartościami zamienionymi na klucze (pogrupowane
        według wartości). Wszystkie wartości muszą być kodowane,
        aby mogły być kluczami w słowniku/zbiorze.
        invertdict(dict(a=1, b=2, c=1)) => {1: ['a', 'c'], 2: ['b']}
    """
    def keysof(V):
        return sorted(K for K in D.keys() if D[K] == V)
    return {V: keysof(V) for V in set(D.values())}

def dflr(cls):
    """
        Klasyczna kolejność przeszukiwania drzewa w cls najpierw w głęb
        potem od lewej do prawej. Cykle nie są możliwe: Python nie
        pozwala na zmianianie __bases__.
    """
    here = [cls]
    for sup in cls.__bases__:
        here += dflr(sup)
    return here

def inheritance(instance):
    """
        Sekwencja kolejności dziedziczenia: w nowym stylu (MRO) lub klasyczna
        (DFLR)
    """
    if hasattr(instance.__class__, '__mro__'):
        return (instance,) + instance.__class__.__mro__
    else:
        return [instance] + dflr(instance.__class__)

def mapattrs(instance, withobject=False, bysource=False):
    """
        Słownik z kluczami dającymi wszystkie odziedziczone atrybuty instancji
        i wartościami dającymi obiekt, po którym zostały odziedziczone.
    """

```

```

withobject: False = usunięcie wbudowanych atrybutów klasy
bysource: True = grupowanie wyniku według obiektów, a nie atrybutów
obsługuje klasy ze slotami wykluczającymi __dict__ w instancjach.

"""

attr2obj = {}
inherits = inheritance(instance)
for attr in dir(instance):
    for obj in inherits:
        if hasattr(obj, '__dict__') and attr in obj.__dict__: # Zobacz
sloty
            attr2obj[attr] = obj
            break
    if not withobject:
        attr2obj = filterdictvals(attr2obj, object)
return attr2obj if not bysource else invertdict(attr2obj)

if __name__ == '__main__':
    print('Zwykłe klasy w 2.x, w nowym stylu w 3.x')
    class A:          attr1 = 1
    class B(A):       attr2 = 2
    class C(A):       attr1 = 3
    class D(B, C):   pass
    I = D()
    print('Py=>%s' % I.attr1)                      # Wyszukiwanie Pythona
== nasze?
    trace(inheritance(I),                  'INH\n')      # [Kolejność
dziedziczenia]
    trace(mapattrs(I),                   'ATTRS\n')     # Atrybuty => źródło
    trace(mapattrs(I, bysource=True), 'OBJS\n')     # Źródło => [atrybuty]
    print('Klasy w nowym stylu w2.x i 3.x')
    class A(object): attr1 = 1                      # "(object)" opcjonalny
w 3.x
    class B(A):          attr2 = 2
    class C(A):          attr1 = 3
    class D(B, C):      pass
    I = D()
    print('Py=>%s' % I.attr1)

```

```

trace(inheritance(I),           'Dziedziczenie\n')
trace(mapattrs(I),             'Atrybuty\n')
trace(mapattrs(I, bysource=True), 'Obiekty\n')

```

W pliku przyjęte jest założenie, że metoda `dir` zwraca wszystkie atrybuty instancji. Następnie każdy atrybut jest wiązany ze swoim źródłem poprzez skanowanie kolejności MRO w przypadku klas w nowym stylu lub kolejności DFLR w przypadku zwykłych klas. Przeszukiwana jest przestrzeń nazw `__dict__` każdego obiektu. W przypadku zwykłych klas kolejność DFLR jest określana poprzez proste skanowanie rekurencyjne. Ostatecznym celem jest symulacja przeszukiwania dziedziczenia w obu modelach klas.

Ten samotestujący kod stosuje swoje narzędzia w opisanym wcześniej diamentowym drzewie wielokrotnego dziedziczenia. Do czytelnego wyświetlania list i słowników wykorzystuje moduł `pprint`. Podstawową metodą jest `pprint.pprint`, a `pformat` zwraca ciąg znaków przeznaczony do wyświetlenia. Kod należy uruchomić w wersji 2.7, aby zobaczyć zarówno klasyczną kolejność DFLR, jak i MRO w nowym stylu. W wersji 3.x dziedziczenie klasy `object` nie jest konieczne i oba testy dają te same wyniki w nowym stylu. Ważne jest to, że atrybut `attr1`, którego wartość jest oznaczona ciągiem `Py=>`, a jego nazwa pojawia się w liście wyników, jest dziedziczony po klasie A w klasycznym wyszukiwaniu, natomiast po klasie C w wyszukiwaniu w nowym stylu:

```

c:\code> py -2 mapattrs.py
Zwykłe klasy w 2.x, w nowym stylu w 3.x
Py=>1
Dziedziczenie
[<__main__.D instance at 0x000000000225A688>,
<class __main__.D at 0x0000000002248828>,
<class __main__.B at 0x0000000002248768>,
<class __main__.A at 0x0000000002248708>,
<class __main__.C at 0x00000000022487C8>,
<class __main__.A at 0x0000000002248708>]

Atrybuty
{['__doc__': <class __main__.D at 0x0000000002248828>,
'__module__': <class __main__.D at 0x0000000002248828>,
'attr1': <class __main__.A at 0x0000000002248708>,
'attr2': <class __main__.B at 0x0000000002248768>}

Obiekty
{<class __main__.A at 0x0000000002248708>: ['attr1'],
<class __main__.B at 0x0000000002248768>: ['attr2'],
<class __main__.D at 0x0000000002248828>: ['__doc__', '__module__']}

Klasy w nowym stylu w 2.x i 3.x
Py=>3
Dziedziczenie

```

```

(<__main__.D object at 0x0000000002257B38>,
<class '__main__.D'>,
<class '__main__.B'>,
<class '__main__.C'>,
<class '__main__.A'>,
<type 'object'>

Atrybuty
{'__dict__': <class '__main__.A'>,
 '__doc__': <class '__main__.D'>,
 '__module__': <class '__main__.D'>,
 '__weakref__': <class '__main__.A'>,
 'attr1': <class '__main__.C'>,
 'attr2': <class '__main__.B'>}

Obiekty
{<class '__main__.A'>: ['__dict__', '__weakref__'],
<class '__main__.B'>: ['attr2'],
<class '__main__.C'>: ['attr1'],
<class '__main__.D'>: ['__doc__', '__module__']}

```

Poniżej przedstawione jest szersze wykorzystanie tych narzędzi. Jest to nasz symulator działający w wersji 3.3 na klasach testowych z poprzedniego rozdziału, zapisanych w pliku *testmixin0.py* (w celu zaoszczędzenia miejsca zostały z niego usunięte niektóre wbudowane nazwy; jak zwykle kod został uruchomiony dla całej listy). Należy zwrócić uwagę, jak pseudoprывatne nazwy X są powiązane z definiującymi je klasami oraz jak klasa *ListInstance* pojawia się w MRO *przed* klasą *object* posiadającą atrybut str, który w innym przypadku zostałby wybrany jako pierwszy. Jak pamiętamy, zmieszanie tej metody było sednem klas programu!

```

c:\code> py -3
>>> from mapattrs import trace, dflr, inheritance, mapattrs
>>> from testmixin0 import Sub
>>> I = Sub()                      # Klasa Sub pochodzi od Super i
ListInstance
>>> trace(dflr(I.__class__))       # Kolejność wyszukiwania w wersji 2.x:
niejawnny obiekt przed listerem!
[<class 'testmixin0.Sub'>,
<class 'testmixin0.Super'>,
<class 'object'>,
<class 'listinstance.ListInstance'>,
<class 'object'>]

```

```

>>> trace(inheritance(I))      # Kolejność wyszukiwania w wersji 3.x (i 2.x
w nowym stylu): najpierw lister
(<testmixin0.Sub object at 0x0000000002974630>,
<class 'testmixin0.Sub'>,
<class 'testmixin0.Super'>,
<class 'listinstance.ListInstance'>,
<class 'object'>
>>> trace(mapattrs(I))
{'_ListInstance__attrnames': <class 'listinstance.ListInstance'>,
'__init__': <class 'testmixin0.Sub'>,
'__str__': <class 'listinstance.ListInstance'>,
...itd...
'data1': <testmixin0.Sub object at 0x0000000002974630>,
'data2': <testmixin0.Sub object at 0x0000000002974630>,
'data3': <testmixin0.Sub object at 0x0000000002974630>,
'szynka': <class 'testmixin0.Super'>,
'mielonka': <class 'testmixin0.Sub'>}
>>> trace(mapattrs(I, bysource=True))
{<testmixin0.Sub object at 0x0000000002974630>: ['data1', 'data2', 'data3'],
<class 'listinstance.ListInstance'>: ['_ListInstance__attrnames',
'__str__'],
<class 'testmixin0.Super'>: ['__dict__', '__weakref__', 'szynka'],
<class 'testmixin0.Sub'>: ['__doc__',
'__init__',
'__module__',
'__qualname__',
'mielonka']}
>>> trace(mapattrs(I, withholdobject=True))
{'_ListInstance__attrnames': <class 'listinstance.ListInstance'>,
'__class__': <class 'object'>,
'__delattr__': <class 'object'>,
...itd...

```

Poniżej przedstawiony jest kod, który można uruchomić w celu przypisania obiektom klas nazw odziedziczonych przez instancję. Jednak aby oszczędzić wzrok użytkownika, trzeba odfiltrować niektóre wbudowane nazwy z podwójnymi znakami podkreślenia!

```
>>> amap = mapattrs(I, withholdobject=True, bysource=True)
```

```

>>> trace(amap)

{<testmixin0.Sub object at 0x000000002974630>: ['data1', 'data2', 'data3'],
 <class 'listinstance.ListInstance': ['_ListInstance__attrnames',
 '_str__'],
 <class 'testmixin0.Super': ['__dict__', '__weakref__', 'szynka'],
 <class 'testmixin0.Sub': ['__doc__',
 '__init__',
 '__module__',
 '__qualname__',
 'mielonka'],
 <class 'object': ['__class__',
 '__delattr__',
 ...itd...
 '__sizeof__',
 '__subclasshook__']}

```

Poniższy kod stanowi uzupełnienie rozważań z poprzedniego rozdziału i płynnie wprowadza nas do następnego podrozdziału. Pokazuje, jak opisany schemat działa również w odniesieniu do atrybutów slotów opartych na klasach. Ponieważ atrybut `__dict__` zawiera zarówno atrybuty zwykłej klasy, jak i indywidualne elementy atrybutów instancji zdefiniowane przez jej listę `__slots__`, atrybuty slotów odziedziczone przez instancję zostaną poprawnie powiązane z klasą implementującą, z której zostały pozyskane, nawet jeśli nie są fizycznie zapisane w atrybutie `__dict__` instancji.

```

# Plik mapattrs-slots.py: test dziedziczenia atrybutu __slots__
from mapattrs import mapattrs, trace

class A(object): __slots__ = ['a', 'b']; x = 1; y = 2
class B(A): __slots__ = ['b', 'c']
class C(A): x = 2
class D(B, C):
    z = 3
    def __init__(self): self.name = 'Robert';
I = D()
trace(mapattrs(I, bysource=True))           # Również: trace(mapattrs(I))

```

Dla jawnie zadeklarowanej klasy w nowym stylu, tak jak w tym przykładzie, wyniki uzyskane w wersjach 2.7 i 3.3 są takie same. Jednak w wersji 3.3 do zbioru dodawana jest dodatkowa wbudowana nazwa. Nazwy odzwierciedlają wszystkie atrybuty odziedziczone przez instancję po klasach zdefiniowanych przez użytkownika, nawet tych zaimplementowanych za pomocą slotów zdefiniowanych w klasach i zapisanych w przestrzeni przydzielonej instancji:

```

c:\code> py -3 mapattrs-slots.py
{<__main__.D object at 0x0000000028988E0>: ['name'],

```

```
<class '__main__.C': ['x'],
<class '__main__.D': ['__dict__',
    '__doc__',
    '__init__',
    '__module__',
    '__qualname__',
    '__weakref__',
    'z'],
<class '__main__.A': ['a', 'y'],
<class '__main__.B': ['__slots__', 'b', 'c']}>
```

Musimy jednak iść dalej, aby lepiej zrozumieć rolę slotów oraz dowiedzieć się, dlaczego funkcja `mapattrs` musi ostrożnie sprawdzać, czy atrybut `__dict__` istnieje, zanim go odczyta.

Przeanalizujmy powyższy kod, aby więcej się o nim dowiedzieć. Kolejnym krokiem w programie listującym z poprzedniego rozdziału może być utworzenie indeksu wynikowego słownika `bysource=True` funkcji `mapattrs` w celu uzyskania atrybutów obiektu podczas przemierzania szkicu drzewa zamiast (a może oprócz) skanowania jego bieżącego fizycznego atrybutu `__dict__`. Prawdopodobnie trzeba będzie wykorzystać metodę `getattr` instancji do pobrania wartości atrybutów, ponieważ niektóre mogą być zaimplementowane jako sloty lub innego rodzaju „wirtualne” atrybuty w ich klasach źródłowych, a pobieranie ich bezpośrednio w klasie nie zwróci wartości instancji. Jeżeli cokolwiek jeszcze tutaj zakoduję, pozbawię czytelnika przyjemności i zniechęzę do tematu następnego podrozdziału.



Moduł `pprint` wykorzystany w tym rozdziale działa w pokazany sposób w wersjach 3.3 i 2.7, ale w wersjach 3.2 i 3.1 zgłasza błąd. Podczas wyświetlania obiektów wywoływany jest wyjątek informujący o błędnej liczbie argumentów. Ponieważ zbyt wiele miejsca poświęciłem opisowi przejściowych defektów Pythona, a poza tym ten problem został rozwiązany w wersjach wykorzystanych w tej książce, sposób obejścia błędu pozostawię jako ćwiczenie dla użytkowników korzystających z wadliwych wersji Pythona. W razie potrzeby można zmienić instrukcję `trace` na zwykłą metodę `print`. Należy pamiętać o *zależności baterii* z rozdziału 1.!

Nowości w klasach w nowym stylu

Oprócz zmian opisanych w poprzednim punkcie (które są, szczerze mówiąc, zbyt akademickie i zaawansowane, aby mogły mieć szczególnie znaczenie dla większości czytelników) klasy w nowym stylu oferują kilka nowych możliwości o bardziej bezpośrednich i praktycznych zastosowaniach, m.in. *sloty*, *właściwości* i *deskryptory*. Kolejne punkty zawierają przegląd tych nowości, które są dostępne w klasach w nowym stylu w Pythonie 2.x oraz we wszystkich klasach w 3.x. Do tej kategorii nowości należą również atrybut `__mro__` i funkcja `super`. Pierwsza została opisana w poprzednim podrozdziale poświęconym zmianom, a druga jest odłożona na koniec rozdziału, ponieważ jest to przypadek wymagający dokładniejszego zbadania.

Sloty: deklaracje atrybutów

Przypisując sekwencję nazw atrybutów tekstowych specjalnemu atrybutowi klasy `__slots__`, można zarówno ograniczać w klasach w nowym stylu zbiór atrybutów, które mogą posiadać instancje klasy, jak również optymalizować wykorzystanie pamięci i potencjalnie przyspieszać działanie programu. Jak się przekonamy, ze slotów należy jednak korzystać tylko w aplikacjach, w których dodatkowa złożoność jest jednoznacznie uzasadniona. Sloty komplikują kod, mogą powodować jego awarie i aby były skuteczne, wymagają uniwersalnego wdrażania.

Podstawy slotów

Aby móc używać slotów, należy przypisać sekwencję nazw specjalnej zmiennej `__slots__` i atrybutowi na najwyższym poziomie instrukcji `class`. Tylko te nazwy, które znajdują się na liście `__slots__`, mogą być atrybutami instancji, którym będzie można przypisywać wartości. Jednak tak jak w przypadku wszystkich nazw w Pythonie, atrybutom instancji o zadanych nazwach, nawet jeżeli są wymienione w zmiennej `__slots__`, muszą zostać przypisane wartości, zanim zaczniemy się do nich odwoływać.

```
>>> class limiter(object):
...     __slots__ = ['age', 'name', 'job']
...
>>> x = limiter()
>>> x.age                                     # Przed użyciem należy przypisać
wartość
AttributeError: age
>>> x.age = 40
>>> x.age
40
>>> x.ape = 1000                                # Nielegalne wywołanie: nazwa
niezdefiniowana w __slots__
AttributeError: 'limiter' object has no attribute 'ape'
```

Funkcjonalność ta ma z założenia służyć wychwytywaniu błędów literowych jak w powyższym przykładzie (wykrywane są przypisania wartości do nazw atrybutów, które nie są wymienione w zmiennej `__slots__`), jak również jako mechanizm optymalizacyjny.

Gdy tworzonych jest wiele instancji, a wymaganych jest tylko kilka atrybutów, wtedy przydzielanie słownika przestrzeni nazw każdemu obiekowi instancji może być kosztowną operacją pod względem wykorzystania pamięci. Aby oszczędzić miejsce, Python nie przydziela słownika każdej *instancji*, tylko rezerwuje w każdej z nich przestrzeń niezbędną do przechowywania wartości każdego atrybutu slotu wraz z odziedziczonimi atrybutami we wspólnej *klasie* na potrzeby zarządzania dostępem do slotów. W ten sposób przyspiesza się wykonywanie kodu, choć zysk jest mniej wyraźny i może być różny w zależności od programu, systemu operacyjnego i wersji Pythona.

Sloty można postrzegać jako pewnego rodzaju wyłąm w dynamicznej naturze Pythona, narzucającej tworzenie wszelkich nazw poprzez przypisania. W rzeczywistości naśladują one język C++, w którym wydajność osiąga się kosztem elastyczności, nawet jeżeli pojawią się niebezpieczeństwo *zakłócenia* działania części programów. Jak zobaczymy, użyciem slotów rzadzi mnóstwo reguł. Zgodnie z oficjalnym podręcznikiem do Pythona slotów *nie* należy używać z wyjątkiem naprawdę szczególnych przypadków. Właściwe korzystanie z nich jest trudne, a one same zgodnie z dokumentacją są zarezerwowane dla rzadkich przypadków

aplikacji wykorzystujących dużą liczbę instancji, gdzie wykorzystanie pamięci jest kwestią krytyczną.

Innymi słowy, jest to kolejna funkcjonalność, której można używać tylko wtedy, gdy jest to naprawdę uzasadnione. Niestety, sloty są wykorzystywane znacznie częściej, niż powinny. Wygląda na to, że ich tajemniczość jest tak pociągająca. Jak zwykle w takich przypadkach najlepszym sprzymierzeńcem jest własna wiedza, dlatego rzućmy teraz okiem na ten temat.



W Pythonie 3.3 dzięki wprowadzeniu modelu słownika ze współdzielonym kluczem zmniejszyły się wymagania dotyczące miejsca dla atrybutów *innych niż sloty*. Słowniki `_dict_` wykorzystywane do przechowywania atrybutów obiektów mogą współdzielić między sobą wewnętrzne miejsce w pamięci, z *kluczami włącznie*. Jest to narzędzie optymalizacyjne zmniejszające niektóre wartości `_slots_`. Według raportów testowych zmiana ta zmniejszała ilość pamięci zajmowanej przez programy obiektowe o 10 - 20%, nieznacznie przyspiesza działanie programów tworzących wiele podobnych obiektów i otwiera drogę do dalszej optymalizacji w przyszłości. Z drugiej strony nie neguje to użycia atrybutu `_slots_` w istniejącym kodzie, który trzeba zrozumieć!

Sloty i słowniki przestrzeni nazw

Pomijając potencjalne korzyści, sloty mogą istotnie komplikować model klas i oparty na nim kod. W rzeczywistości niektóre instancje ze slotami mogą w ogóle nie mieć słownika atrybutów przestrzeni nazw `_dict_`, a inne mogą mieć atrybuty danych, których ten słownik nie zawiera. Żeby było jasne: jest to *duża niekompatybilność* z modelem tradycyjnych klas i może ona skomplikować każdy kod odwołujący się w podstawowy sposób do atrybutów, a nawet powodować awarie niektórych programów.

Na przykład programy, które tworzą listy atrybutów lub odwołują się do nich za pomocą nazw, mogą wymagać użycia bardziej neutralnych pamięciowo interfejsów niż `_dict_`, aby można było użyć slotów. Ponieważ dane instancji mogą zawierać nazwy na poziomie klas, na przykład sloty — czy to dodatkowo, czy zamiast słownika przestrzeni nazw — oba źródła atrybutów mogą wymagać sprawdzania pod kątem kompletności.

Sprawdźmy, co to oznacza dla kodu, a przy okazji dowiedzmy się więcej o slotach. Przede wszystkim, jeśli zostały użyte sloty, instancje nie będą miały słownika atrybutów, a Python do zaalokowania slotów dla atrybutów wykorzysta *deskryptory* klas, opisane w dalszej części książki. Oto przykład dla Pythona 3.x (oraz 2.x przy użyciu klas w nowym stylu):

```
>>> class C:                                         # W wersji 2.x (tylko)
    wymagane "(object)"

    ...      __slots__ = ['a', 'b']                  # __slots__ oznacza, że
    nie ma __dict__

    ...

>>> X = C()
>>> X.a = 1
>>> X.a
1
>>> X.__dict__
AttributeError: 'C' object has no attribute '__dict__'
```

Wciąż jednak można odczytywać i przypisywać wartości atrybutom według nazw, wykorzystując neutralne pamięciowo narzędzia, takie jak `getattr`, `setattr` (które sięgają poza atrybut `__dict__` instancji, dzięki czemu obejmują nazwy na poziomie klas, na przykład sloty) oraz `dir` (które zbiera wszystkie odziedziczone nazwy z całego drzewa klas):

```
>>> getattr(X, 'a')
1
>>> setattr(X, 'b', 2)                      # Ale getattr() i setattr() będą
                                               działać
>>> X.b
2
>>> 'a' in dir(X)                         # dir() również znajduje atrybuty
                                               w slotach
True
>>> 'b' in dir(X)
True
```

Bez słownika przestrzeni nazw nie jest możliwe przypisanie nowych nazw atrybutów w instancji, jeśli nie są zadeklarowane na liście slotów:

```
>>> class D:                                # Aby ten sam wynik uzyskać w wersji 2.x,
                                               należy użyć D(object)
    __slots__ = ['a', 'b']
    def __init__(self):
        self.d = 4                          # Nie można dodawać nazw niezdefiniowanych
                                               w __dict__
>>> X = D()
AttributeError: 'D' object has no attribute 'd'
```

Dodatkowe atrybuty można jednak obsługiwać, deklarując nazwę `__dict__` w atrybucie `__slots__`, co pozwoli na wykorzystanie w instancjach słownika przestrzeni nazw.

```
>>> class D:
...     __slots__ = ['a', 'b', '__dict__'] # Deklarujemy __dict__ jako jeden
                                               ze slotów
...     c = 3                            # Atrybuty klasy działają bez
                                               zmian
...     def __init__(self): self.d = 4      # d zostaje zapisane w __dict__, a
                                               w __slots__
...
>>> X = D()
>>> X.d
4
>>> X.c
```

```
>>> X.a          # Wszystkie atrybuty instancji pozostają
niezdefiniowane do chwili przypisania wartości

AttributeError: a

>>> X.a = 1

>>> X.b = 2
```

W tym przypadku wykorzystane będą *obydwa* mechanizmy zapisu atrybutów. To sprawia, że atrybut `__dict__` staje się zbyt ograniczony w kodzie, w którym sloty mają być traktowane jak dane instancji, ale generyczne narzędzia, takie jak `getattr`, będą je traktować jako jednolity zestaw atrybutów.

```
>>> X.__dict__           # Niektóre obiekty posiadają
__dict__ oraz __slots__
{'d': 4}                  # getattr() odczyta wartość
atrybutu dowolnego rodzaju

>>> X.__slots__
['a', 'b', '__dict__']

>>> setattr(X, 'a',), setattr(X, 'c'), setattr(X, 'd')      # Pobranie
wszystkich trzech form

(1, 3, 4)
```

Metoda `dir` zwraca jednak także *odziedziczone* atrybuty, których może być zbyt dużo w niektórych kontekstach. Zawiera również metody na poziomie klas, a nawet wartości domyślne wszystkich obiektów. W kodzie, w którym trzeba wymienić tylko atrybuty instancji, może być w zasadzie nadal wymagane wyraźne dopuszczenie obu form przechowywania danych. Pierwsze naturalne rozwiązanie może wyglądać następująco:

```
>>> for attr in list(X.__dict__) + X.__slots__:
# Błąd...
...     print(attr, '=>', getattr(X, attr))
```

Ponieważ każdą z form można pominąć, można napisać bardziej poprawny kod, jak niżej, wykorzystując metodę `getattr`, aby dopuścić wartości domyślne. Jest to poprawne, choć niedokładne rozwiązanie, opisane w następnym podrozdziale.

```
>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
...     print(attr, '=>', getattr(X, attr))

d => 4
a => 1                         # Mniejszy błąd...
b => 2
__dict__ => {'d': 4}
```

Wiele slotów w klasach nadrzędnych

Opisany wyżej kod działa poprawnie w szczególnym przypadku, ale ogólnie *nie jest całkowicie ścisły*. Konkretnie mówiąc, obsługuje wyłącznie nazwy zdefiniowane w `__slots__` na *najniższym* poziomie drzewa dziedziczenia, ale przecież listy slotów mogą pojawiać się więcej niż raz w drzewie klas. Oznacza to, że brak nazwy w najniższej liście `__slots__` nie wyklucza jej istnienia na wyższych tego rodzaju listach. Ponieważ nazwy slotów stają się atrybutami na

poziomie klas, instancje pozyskują nazwy wszystkich slotów w całym drzewie, zgodnie ze zwykłą regułą dziedziczenia.

```
>>> class E:  
...     __slots__ = ['c', 'd']          # Klasa nadrzędna posiada sloty  
>>> class D(E):  
...     __slots__ = ['a', '__dict__']      # Podobnie jej klasa potomna  
...  
>>> X = D()  
>>> X.a = 1; X.b = 2; X.c = 3        # Instancja jest połączeniem tych  
klas  
>>> X.a, X.c  
(1, 3)
```

Sprawdzając wyłącznie listę odziedziczonych slotów, nie znajdziemy slotów zdefiniowanych wyżej w drzewie klas:

```
>>> E.__slots__                      # Ale sloty nie są dziedziczone z  
klas nadrzędnych  
['c', 'd']  
>>> D.__slots__  
['a', '__dict__']  
>>> X.__slots__                      # Instancja dziedziczy sloty  
wyłącznie z jej klasy  
['a', '__dict__']  
>>> X.__dict__                       # I posiada własny słownik  
__attr__  
{'b': 2}  
>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__',[]):  
...     print(attr, '=>', getattr(X, attr))  
...  
b => 2                                # Brakuje slotów z klasy  
nadrzędnej!  
a => 1  
__dict__ => {'b': 2}  
>>> dir(X)                            # dir() wyświetla wszystkie nazwy  
[...pominięta część wyniku... 'a', 'b', 'c', 'd']
```

Innymi słowy, w kwestii ogólnego wyświetlania atrybutów instancji jeden atrybut `__slots__` nie zawsze wystarcza — potencjalnie trzeba wobec nich stosować pełną procedurę przeszukiwania drzewa dziedziczenia. Opisany wcześniej plik `mapattrs-slots.py` zawiera inny przykład slotów pojawiających się w wielu klasach nadrzędnych. Jeżeli kilka klas w drzewie posiada własne

atrybuty `__slots__`, w generycznych programach trzeba implementować inne procedury wyświetlania atrybutów, o czym będzie mowa w następnym podrozdziale.

Generyczna obsługa slotów i innych „wirtualnych” atrybutów

W tym miejscu warto wrócić do opisu opcji slotów i kodu *lister.py* wyświetlającego mieszane klasy. Jest to doskonały przykład generycznego programu, w którym należy troszczyć się o sloty. Narzędzia, które w generyczny sposób wyświetlają atrybuty instancji, muszą uwzględniać sloty, jak również inne „wirtualne” atrybuty instancji, takie jak *właściwości* i *deskryptory* opisane w dalszej części książki. Pojęcia te również istnieją w klasach, ale na żądanie mogą dostarczać innych wartości atrybutów instancji. Sloty są najbardziej ukierunkowane na dane, ale reprezentują szerszą kategorię.

Tego rodzaju atrybuty wymagają integracyjnego podejścia i specjalnego traktowania, ale najlepiej jest ich unikać. Ostatnia opcja nie będzie jednak odpowiednia, jeżeli jakiś programista zacznie używać slotów. Atrybuty instancji na poziomie klasy naprawdę wymagają nowej definicji terminu *dane instancji* jako lokalnie zapisanych wszystkich odziedziczonych atrybutów lub ich podzbioru.

Na przykład niektóre programy mogą klasyfikować nazwy slotów jako atrybuty *klasy*, a nie instancji. Takie atrybuty nie istnieją w słownikach przestrzeni nazw instancji. Ewentualnie, jak pokazałem wcześniej, programy mogą być bardziej integracyjne i wykorzystywać metodę `dir` do pobierania nazw wszystkich odziedziczonych atrybutów oraz metodę `getattr` do pobierania odpowiadających im wartości w instancji, bez względu na ich fizyczną lokalizację lub implementację. Jeżeli trzeba obsługiwać sloty jako dane instancji, można zastosować przedstawiony niżej, prawdopodobnie najlepszy sposób:

```
>>> class Slotful:
    __slots__ = ['a', 'b', '__dict__']
    def __init__(self, data):
        self.c = data

>>> I = Slotful(3)
>>> I.a, I.b = 1, 2
>>> I.a, I.b, I.c                                # Zwykłe pobranie
atrybutów
(1, 2, 3)
>>> I.__dict__                                     # Pamięć __dict__ i
slots
{'c': 3}
>>> [x for x in dir(I) if not x.startswith('__')]
['a', 'b', 'c']                                    # __dict__ jest tylko
jednym źródłem atrybutów
3
>>> getattr(I, 'c'), getattr(I, 'a')               # dir+getattr jest
szersze niż __dict__
(3, 1)                                              # Dotyczy slotów,
właściwości i deskryptorów
```

```

>>> for a in (x for x in dir(I) if not x.startswith('__')):
    print(a, getattr(I, a))

a 1
b 2
c 3

```

W tym modelu `dir/getattr` wciąż można wiązać atrybuty z ich źródłami pochodzenia i w razie potrzeby bardziej selektywnie je filtrować według źródła lub typu poprzez skanowanie MRO, tak jak robiliśmy to wcześniej w plikach `mapattrs.py` i `mapattrs-slots.py`, gdzie stosowane były sloty. Dodatkowo takie narzędzia i zasady obsługi slotów potencjalnie można automatycznie stosować w odniesieniu do właściwości i deskryptorów, choć atrybuty te w porównaniu ze slotami bardziej są jawnie wyliczonymi wartościami, a mniej wyraźnie danymi związanymi z instancjami.

Należy również pamiętać, że nie jest to tylko kwestia narzędzi. Atrybuty instancji opartych na klasach, takie jak sloty, również mają wpływ na tradycyjne kodowanie poznanej w rozdziale 30. metody `__setattr__` przeciążającej operator. Ponieważ sloty i kilka innych atrybutów nie są przechowywane w słowniku `__dict__` instancji, a nawet może nie być ich wcale, klasy w nowym stylu muszą zazwyczaj wykonywać operacje przypisywania wartości atrybutom poprzez kierowanie ich do klasy `object`. W praktyce z tego powodu powyższa metoda może wyglądać zupełnie inaczej w niektórych klasach zwykłych i w nowym stylu.

Zasady używania slotów

Deklaracje slotów mogą pojawiać się w wielu klasach w drzewie, ale podlegają pewnej liczbie reguł, które dość trudno jest zrozumieć, dopóki nie pozna się implementacji slotów jako deskryptorów na poziomie klas dla każdej nazwy slotu dziedziczonego przez instancje, w których rezerwowana jest zarządzana przestrzeń (deskryptor jest zaawansowanym narzędziem, które zbadamy szczegółowo w ostatniej części książki).

- *Sloty w klasach podrzędnych są bezużyteczne, jeżeli nie ma ich w klasach nadrzędnych.* Jeżeli klasa nadrzędna nie ma atrybutu `__slots__`, wtedy instancja atrybutu `__dict__` utworzona dla klasy nadrzędnej zawsze będzie dostępna, co oznacza, że atrybut `__slots__` w klasie podrzędnej w dużej mierze nie będzie potrzebny. Klasa podrzędna wciąż zarządza swoimi slotami, ale w żaden sposób nie wylicza ich wartości i nie pomija słownika — głównego powodu stosowania slotów.
- *Sloty w klasach nadrzędnych są bezużyteczne, jeżeli nie ma ich w klasach podrzędnych.* Analogicznie, ponieważ znaczenie deklaracji `__slots__` jest ograniczone do klasy, w której została użyta, klasy podrzędne tworzą instancje słownika `__dict__`, jeżeli nie definiują `__slots__`. Z tego powodu tworzenie atrybutu `__slots__` w klasie nadrzędnej jest w dużej mierze bezcelowe.
- *Redefinicja sprawia, że sloty w klasach nadrzędnych są bezużyteczne.* Jeżeli klasa definiuje tę samą nazwę slotu co jej klasa nadrzędna, wtedy powtórna definicja ukrywa slot w klasie nadrzędnej podczas zwykłego dziedziczenia. Do nazwy slotu zdefiniowanego w klasie nadrzędnej można odwoływać się jedynie poprzez pobieranie jego deskryptora bezpośrednio z klasy nadrzędnej.
- *Sloty uniemożliwiają stosowanie domyślnych ustawień na poziomie klasy.* Ponieważ sloty są implementowane jako deskryptory na poziomie klas (wraz z przestrzenią dla każdej instancji), nie można wykorzystywać atrybutów klasy o takich samych nazwach do wskazywania wartości domyślnych, tak jak to się robi w przypadku zwykłych atrybutów instancji. Przypisanie tej samej nazwy w klasie skutkuje nadpisaniem deskryptora slotu.
- *Sloty i atrybut `__dict__`.* Jak wspomniałem wcześniej, atrybut `__slots__` wyklucza zarówno korzystanie z atrybutu `__dict__`, jak również przypisywanie nazw, których nie ma na liście, chyba że atrybut `__dict__` jest również jawnie wymieniony.

Ostatnią regułę widzieliśmy już w akcji, a trzecią regułę ilustruje opisany wcześniej plik *mapattrs-slots.py*. Łatwo jest pokazać, jak nowe reguły przekładają się na rzeczywisty kod. Co ciekawe, słownik przestrzeni nazw jest tworzony wtedy, gdy dowolna klasa w drzewie pomija sloty, tym samym negując korzyści związane z optymalizacją pamięci:

```
>>> class C: pass                                # Reguła 1: sloty w klasie
podzielonej, ale nie w nadzędnej

>>> class D(C): __slots__ = ['a']                 # Utworzenie słownika instancji
bez slotów

>>> X = D()                                       # Nazwa slotu wciąż jest
zarządzana w klasie

>>> X.a = 1; X.b = 2

>>> X.__dict__
{'b': 2}

>>> D.__dict__.keys()
dict_keys([... 'a', '__slots__', ...])

>>> class C: __slots__ = ['a']                     # Reguła 2: sloty w klasie
nadzędnej, ale nie w podzielnej

>>> class D(C): pass                            # Utworzenie słownika instancji
bez slotów

>>> X = D()                                       # Nazwa slotu wciąż jest
zarządzana w klasie

>>> X.a = 1; X.b = 2

>>> X.__dict__
{'b': 2}

>>> C.__dict__.keys()
dict_keys([... 'a', '__slots__', ...])

>>> class C: __slots__ = ['a']                     # Reguła 3: dostępny tylko
najniższy slot

>>> class D(C): __slots__ = ['a']

>>> class C: __slots__ = ['a']; a = 99            # Reguła 4: brak domyślnych
ustawień na poziomie klasy

ValueError: 'a' in __slots__ conflicts with class variable
```

Innymi słowy, sloty, oprócz tego, że potencjalnie uniemożliwiają działanie programu, to aby były efektywne, wymagają *uniwersalnego i starannego wdrażania*. Ponieważ w odróżnieniu od właściwości (opisanych w następnym podrozdziale) nie wyliczają one wartości dynamicznie, w dużej mierze są bezużyteczne, chyba że stosuje się je w każdej klasie w drzewie i ostrożnie definiuje się tylko nazwy nowych slotów, które nie są zdefiniowane w innych klasach. Jest to funkcjonalność typu „wszystko lub nic” — niefortunna właściwość współdzielona przez opisane w dalszej części rozdziału wywołanie *super*.

```
>>> class C: __slots__ = ['a']                   # Założone uniwersalne użycie z
innymi nazwami

>>> class D(C): __slots__ = ['b']
```

```

>>> X = D()
>>> X.a = 1; X.b = 2
>>> X.__dict__
AttributeError: 'D' object has no attribute '__dict__'
>>> C.__dict__.keys(), D.__dict__.keys()
(dict_keys([... 'a', '__slots__', ...]), dict_keys([... 'b', '__slots__', ...]))

```

Powysze reguły — oprócz innych dotyczących słabych odwoła, pominiętych tutaj ze względu na brak miejsca — są po części powodem, dla którego stosowanie slotów nie jest zalecane, z wyjątkiem patologicznych przypadków, w których ważne jest zmniejszenie zajmowanej przestrzeni. Jednak nawet wtedy niebezpieczne skomplikowania lub uniemożliwiania działania kodu powinno być wystarczającym powodem do dokładnego rozważania kompromisu. Sloty nie tylko wymagają niemal patologicznego rozmieszczenia w całej platformie, ale także mogą zakłócać działanie sprawdzonych narzędzi.

Przykład stosowania slotów: klasa ListTree i funkcja mapattrs

Przeanalizujmy bardziej praktyczny przykład efektów użycia slotów, jakie wynikają z pierwszej reguły opisanej w poprzednim podrozdziale. Klasa `ListTree`, opisana w rozdziale 31., *nie ulega awarii* po zmieszaniu jej z klasą definiującą atrybut `__slots__`, mimo że skanuje ona słowniki przestrzeni nazw instancji. Pomimo braku slotów w klasie `ListTree` instancja i tak będzie miała atrybut `__dict__`, a więc nie zgłosi wyjątku podczas odczytywania lub indeksowania. Na przykład oba poniższe kody wyświetlają wyniki bez błędów. W drugim przypadku można również przypisywać wartości nazwom, których nie ma w liście slotów, jako atrybutom instancji, z wymaganymi przez klasę nadziedną włącznie:

```

class C(ListTree): pass
X = C()                                     # OK: __slots__ nie jest używany
print(X)
class C(ListTree): __slots__ = ['a', 'b']    # OK: klasa nadziedna tworzy
__dict__
X = C()
X.c = 3
print(X)                                     # Wyświetla atrybut c z klasy X
oraz atrybuty a i b z klasy C

```

Poniższe klasy również poprawnie wyświetlają wyniki. Każda klasa bez slotów, jak `ListTree`, tworzy instancję `__dict__`, więc można bezpiecznie założyć jej istnienie:

```

class A: __slots__ = ['a']                      # Oba OK zgodnie z pierwszą regułą
class B(A, ListTree): pass
class A: __slots__ = ['a']
class B(A, ListTree): __slots__ = ['b']        # Wyświetla atrybut b klasy B i
atrybut a klasy A

```

Choć sloty w klasach podrzędnych są bezużyteczne, jest to pozytywny efekt uboczny stosowania klas takich jak `ListTree` (i jej poprzedniczki opisanej w rozdziale 28.). Zazwyczaj jednak niektóre narzędzia wymagają przechwytywania wyjątków, gdy nie ma atrybutu `__dict__` lub wykorzystywane są metody `hasattr` i `getattr` do sprawdzania lub nadawania domyślnych

ustawień, jeśli stosowanie slotów wyklucza sprawdzanie słownika przestrzeni nazw w obiektach instancji.

Teraz powinno być jasne, dlaczego opisany wcześniej w tym rozdziale program *mapattrs.py* musi sprawdzać obecność atrybutu `__dict__` przed odwołaniem się do niego. Obiekty instancji utworzone z klas zawierających atrybut `__slots__` nie mają tego atrybutu. W rzeczywistości, jeżeli użyje się wyróżnionego niżej alternatywnego wiersza, funkcja `mapattrs` zgłosi wyjątek przy próbie znalezienia nazwy atrybutu w instancji znajdującej się na początku sekwencji dziedziczenia:

```
def mapattrs(instance, withobject=False, bysource=False):
    for attr in dir(instance):
        for obj in inherits:
            if attr in obj.__dict__:                      # Może zgłosić błąd, jeżeli
                użyty jest __slots__
            >>> class C: __slots__ = ['a']
            >>> X = C()
            >>> mapattrs(X)
            AttributeError: 'C' object has no attribute '__dict__'
```

Każdy z poniższych wierszy rozwiązuje problem i umożliwia obsługę slotów w narzędziu. Pierwszy nadaje domyślne ustawienia. Drugi wiersz jest bardziej rozbudowany, ale jego intencja jest wyraźniej widoczna.

```
if attr in getattr(obj, '__dict__', {}):
    if hasattr(obj, '__dict__') and attr in obj.__dict__:
```

Jak wspomniałem wcześniej, niektóre narzędzia mogą korzystać na wiązaniu w opisany sposób wyników metody `dir` z obiektami w MRO zamiast używania ogólnego skanowania atrybutu `__dict__` instancji. Bez tego bardziej integracyjnego podejścia atrybuty zaimplementowane przez narzędzia na poziomie klas, na przykład sloty, nie będą raportowane jako dane instancji. Niekoniecznie zwalnia to tego rodzaju narzędzia od dopuszczenia brakującego `__dict__` w instancji!

Co z szybkością slotów?

Choć sloty przede wszystkim optymalizują wykorzystanie pamięci, ich wpływ na szybkość działania programu jest mniej oczywisty. Poniżej przedstawiony jest prosty skrypt testowy wykorzystujący techniki `timeit` opisane w rozdziale 21. W obu modelach, ze slotami i bez (słownik instancji), tworzonych jest 1000 instancji, a w każdej z nich są przypisywane i odczytywane wartości czterech atrybutów. Operacje te są wykonywane 1000 razy. W obu modelach wybierany jest najlepszy wynik z trzech przebiegów, przy czym w każdym wykonywanych jest 8 milionów operacji na atrybutach.

```
# Plik slots-test.py
from __future__ import print_function
import timeit
base = """
Is = []
for i in range(1000):
```

```

X = C()
X.a = 1; X.b = 2; X.c = 3; X.d = 4
t = X.a + X.b + X.c + X.d
Is.append(X)

"""
stmt = """
class C:
    __slots__ = ['a', 'b', 'c', 'd']
""" + base
print('Sloty =>', end=' ')
print(min(timeit.repeat(stmt, number=1000, repeat=3)))
stmt = """
class C:
    pass
""" + base
print('Bez slotów=>', end=' ')
print(min(timeit.repeat(stmt, number=1000, repeat=3)))

```

Na moim laptopie z zainstalowanymi wersjami Pythona 3.3 i 2.7 uzyskane najlepsze czasy dowodzą, że sloty są nieco szybsze w wersjach 3.x i 2.x. Niewiele to jednak mówi o ilości wykorzystywania pamięci, jak również może się zmienić w przyszłości.

```

C:\code> py -3 slots-test.py
Sloty => 0.7780903942045899
Bez slotów=> 0.9888108080898417

C:\code> py -2 slots-test.py
Sloty => 0.615521153591
Bez slotów=> 0.766582559582

```

Więcej ogólnych informacji o slotach znajduje się w standardowym zestawie podręczników do Pythona. Warto również zapoznać się z opisem dekoratora `Private` w rozdziale 39. Jest to przykład naturalnego wykorzystania atrybutów opartych na pamięci `__slots__` i `__dict__` poprzez użycie delegacji i neutralnych pamięciowo narzędzi dostępowych, takich jak metoda `getattr`.

Właściwości klas: dostęp do atrybutów

Kolejnym rozszerzeniem są *właściwości* — sposób definiowania w klasach w nowym stylu wywoływanych automatycznie metod służących do dostępu lub przypisania do atrybutów instancji. Jest to funkcjonalność podobna do właściwości (zwanych też *getterami* i *setterami*) w innych językach, na przykład Java lub C++. W Pythonie jednak najlepiej jest używać jej oszczędnie jako sposobu na uzyskiwanie dostępu do atrybutów w miarę zmieniających się

wymagań. W razie potrzeby można stosować właściwości, które umożliwiają dynamiczne obliczanie wartości atrybutów bez konieczności wywoływania metod w punkcie dostępu.

Choć właściwości nie obsługują celów kierowania generycznych atrybutów, przynajmniej tych specyficznych, stanowią alternatywę dla wielu wykorzystywanych obecnie zastosowań metod przeciążania `__getattr__` oraz `__setattr__`, które omawialiśmy w rozdziale 30. Właściwości mają podobny efekt do tych dwóch metod, jednak mają dodatkowe wywołanie metody zarezerwowane jedynie dla dostępu do zmiennych, które wymagają dynamicznego obliczenia. Dostęp do nazw innych elementów niż właściwości odbywa się w zwykły sposób bez dodatkowych wywołań. Metoda `__getattr__` jest wywoływana tylko dla *niezdefiniowanych* nazw, natomiast metoda `__setattr__` jest wywoływana przy przypisywaniu wartości *dowolnemu* atrybutowi.

Właściwości i sloty są ze sobą powiązane, ale służą do różnych celów. Tak samo implementują atrybuty instancji, które nie są fizycznie zapisywane w słownikach przestrzeni nazw instancji. Są to swego rodzaju „wirtualne” atrybuty bazujące na *deskryptorach* atrybutów na poziomie klasy. Sloty zarządzają pamięcią instancji, natomiast właściwości przechwytyują dostęp i umożliwiają dowolne wyliczanie wartości. Ponieważ narzędzie implementujące deskryptor, na którym się opierają, jest zbyt zaawansowane, aby je tu opisać, właściwości i deskryptory zostały dokładnie przedstawione w rozdziale 38.

Podstawy właściwości

Mówiąc w skrócie, właściwość jest rodzajem obiektu przypisanego do nazwy atrybutu klasy. Jest generowany przez wywołanie wbudowanej funkcji `property` z trzema metodami (programami obsługi dla operacji pobierania, ustawiania oraz usuwania), a także łańcuchem znaków dokumentacji. Jeśli jakiś argument zostanie przekazany jako `None` lub pominięty, operacja ta nie jest obsługiwana.

Właściwości są zazwyczaj przypisywane na najwyższym poziomie instrukcji `class` (na przykład `name = property(...)`). Krok ten automatyzuje się za pomocą specjalnego znaku @, opisanego w dalszej części rozdziału. Jeśli właściwość przypisze się w taki sposób, dostępy do samego atrybutu klasy (na przykład `obj.name`) są automatycznie przekierowywane do jednej z metod akcesorów przekazanej do funkcji `property`.

Metoda `__getattr__` pozwala na przykład przechwytywać referencje do niezdefiniowanych atrybutów zarówno w klasach zwykłych, jak i w nowym stylu:

```
>>> class classic:  
...     def __getattr__(self, name):  
...         if name == 'age':  
...             return 40  
...         else:  
...             raise AttributeError  
...  
>>> x = classic()  
>>> x.age  
# Wykonuje __getattr__  
40  
>>> x.name  
# Wykonuje __getattr__  
AttributeError
```

Poniżej znajduje się ten sam przykład, jednak tym razem zostały wykorzystane właściwości. Należy zwrócić uwagę, że właściwości działają z dowolnymi klasami, ale w przypadku Pythona 2.x wymagają jawnej deklaracji dziedziczenia po klasie `object`, aby poprawnie przechwytywały przypisania wartości do atrybutów (jeżeli się o tym zapomni, nie pojawi się żaden komunikat o błędzie, ale niepostrzeżenie właściwość zostanie nadpisana nowymi danymi):

```
>>> class newprops(object):          # W wersji 2.x klasa object jest
wymagana dla metod przypisania
...     def getage(self):
...         return 40
...     age = property(getage, None, None, None) # Operacje get, set, del,
dokumentacja lub @
...
>>> x = newprops()
>>> x.age                         # Wykonuje getage
40
>>> x.name                         # Normalne pobranie
AttributeError: newprops instance has no attribute 'name'
```

Dla niektórych zadań programistycznych właściwości mogą być mniej skomplikowane oraz szybsze do wykonania od tradycyjnych technik. Kiedy na przykład dodamy obsługę *przypisania* atrybutów, właściwości staną się bardziej atrakcyjne — do wpisania mamy mniej kodu, a żadne dodatkowe wywołania metod nie są konieczne dla przypisania do atrybutów, których nie chcemy obliczać dynamicznie.

```
>>> class newprops(object):          # W wersji 2.x klasa object jest
wymagana dla metod przypisania
...     def getage(self):
...         return 40
...     def setage(self, value):
...         print('ustawienie wieku: %s' % value)
...         self._age = value
...     age = property(getage, setage, None, None)
...
>>> x = newprops( )
>>> x.age                         # Wykonuje getage
40
>>> x.age = 42                      # Wykonuje setage
ustawienie wieku: 42
>>> x._age                         # Normalne pobranie; nie ma
wywołania getage
42
```

```

>>> x.age                                # Wykonuje getage
40

>>> x.job = 'instruktor'                  # Normalne przypisanie; nie ma
wywołania setage

>>> x.job                                # Normalne pobranie; nie ma
wywołania getage

'instruktor'

Odpowiednik tego kodu wykorzystujący przeciążanie operatorów wymaga kilku dodatkowych
wywołań przypisania niezarządzanych atrybutów i w celu uniknięcia pętli do ich obsługi
wymaga zastosowania słownika (a w klasach w nowym stylu wywołania metody __setattr__
klasy nadzędnej, aby lepiej obsługiwać „wirtualne” atrybuty, takie jak sloty i właściwości
zakodowane w innych klasach):

>>> class classic:
...     def __getattr__(self, name):          # Przy niezdefiniowanej
referencji
...         if name == 'age':
...             return 40
...
...         else:
...             raise AttributeError
...
...     def __setattr__(self, name, value):    # Przy wszystkich przypisaniach
...         print('ustawienie: %s %s' % (name, value))
...
...         if name == 'age':
...             self.__dict__['_age'] = value # Lub object.__setattr__()
...
...         else:
...             self.__dict__[name] = value
...
...
...
>>> x = operators()
>>> x.age                                # Wykonuje __getattr__
40

>>> x.age = 41                            # Wykonuje __setattr__
ustawienie: age 41

>>> x._age                               # Zdefiniowane: nie ma wywołania
__getattr__
41

>>> x.age                                # Wykonuje __getattr__
40

>>> x.job = 'instruktor'                 # Wykonuje znowu __setattr__

```

```
>>> x.job # Zdefiniowane: nie ma wywołania  
__getattr__
```

Właściwości wydają się w tym przypadku lepszym rozwiązaniem. Jednak istnieją sytuacje, gdy lepszym rozwiązaniem są metody `__getattr__` oraz `__setattr__`, na przykład w celu zaimplementowania bardziej dynamicznych czy uniwersalnych interfejsów od tych, które mogą być zaimplementowane przez właściwości.

W wielu przypadkach nie da się na etapie tworzenia kodu klasy ustalić zbioru atrybutów, jaki ma być obsługiwany, może on nawet nie istnieć w jakiejkolwiek namacalnej formie (na przykład kiedy *deleguje* się referencje do dowolnych metod do obiektów opakowanych czy osadzonych w sposób uniwersalny). W takich przypadkach ogólny mechanizm obsługi atrybutów `__getattr__` czy `__setattr__` z przekazaną nazwą atrybutu może być lepszym wyjściem. Ponieważ ten mechanizm potrafi sobie również radzić z prostszymi przypadkami, właściwości są w dużej mierze rozszerzeniem opcjonalnym, choć preferowanym przez programistów, bo pozwalającym uniknąć dodatkowych wywołań przy przypisaniach.

Więcej szczegółów na temat wykorzystania obydwu metod dynamicznej obsługi atrybutów można znaleźć w rozdziale 38. w ostatniej części książki. Znajdziemy tam między innymi zastosowanie *składni dekoratora funkcji* wykorzystującej znak @, która zostanie wprowadzona w dalszej części niniejszego rozdziału. Jest to odpowiednik i automatyczna alternatywa do ręcznego przypisywania w zakresie klas:

```
class properties(object):  
    @property  
        # Kodowanie właściwości za pomocą  
    dekoratorów – patrz opis niżej  
  
    def age(self):  
        ...  
  
        @age.setter  
    def age(self, value):  
        ...
```

Aby objąć składnię dekoratorów, przejdźmy do kolejnego tematu.

Narzędzia atrybutów: `__getattribute__` i deskryptory

Metoda przeciążania `__getattribute__`, dostępna jedynie dla klas w nowym stylu, pozwala klasom na przechwytywanie *wszystkich* referencji do atrybutów, nie tylko referencji niezdefiniowanych. Jej użycie jest również znacznie bardziej skomplikowane w porównaniu do `__getattr__` i podatne na zapętlenia, podobnie do `__setattr__`, choć w inny sposób.

Oprócz metod przeciążania właściwości i operatorów Python oferuje koncepcję *deskryptorów* atrybutów. Są to klasy wyposażone w metody `__set__` i `__get__` przypisane atrybutom klasy i dziedziczone przez instancje. Klasy te przechwytyują żądania odczytu i zapisu atrybutu. Poniżej przedstawiony jest przykład jednego z najprostszych dekoratorów, z jakimi się można spotkać:

```
>>> class AgeDesc(object):  
        def __get__(self, instance, owner): return 40  
        def __set__(self, instance, value): instance._age = value  
  
>>> class descriptors(object):  
        age = AgeDesc()
```

```
>>> x = descriptors()
>>> x.age                                         # Uruchamia AgeDesc.__get__
40
>>> x.age = 42                                     # Uruchamia AgeDesc.__set__
>>> x._age                                         # Zwykłe pobranie, bez wywołania
AgeDesc
42
```

Deskryptory mają dostęp do stanu własnych instancji, jak również do klas klienckich. Są w pewnym sensie uogólnioną koncepcją właściwości, a raczej właściwości stanowią uproszczony sposób definicji specyficznego typu deskryptora — mówiąc dokładniej: takiego deskryptora, który uruchamia wskazane funkcje w reakcji na próby dostępu. Deskryptory są również wykorzystywane do implementacji slotów, które omówiliśmy wcześniej.

Właściwości, metoda `__getattribute__` oraz deskryptory są zagadnieniami dość zaawansowanymi, na razie więc odłożymy ich analizę. Szczegóły dotyczące tych technik oraz pogłębienie tematu właściwości można znaleźć w rozdziale 38. w ostatniej części książki. Zostaną również wykorzystane w przykładach w rozdziale 39. oraz w opisie ich wpływu na dziedziczenie w rozdziale 40.

Inne zmiany i rozszerzenia klas

Jak wspomniałem wcześniej, opis wbudowanej funkcji `super` — dodatkowego ważnego rozszerzenia klasy w nowym stylu opartego na MRO — został odłożony na koniec rozdziału. Zanim do niego dotrzymy, poznajmy dodatkowe zmiany i rozszerzenia, które niekoniecznie dotyczą klas w nowym stylu, ale zostały wprowadzone mniej więcej w tym samym czasie. Są to m.in. metody statyczne, metody klasy i deskryptory.

Większość zmian i dodatkowych funkcji klas w nowym stylu wpasowuje się w koncepcję typów będących klasami, o której wspominaliśmy wcześniej w tym rozdziale, ponieważ ta koncepcja oraz klasy w nowym stylu w ogólności zostały wprowadzone w połączeniu z wyeliminowaniem sytuacji istnienia dwóch modeli klas od Pythona 2.2 wzwyż. Jak mieliśmy okazję się przekonać od Pythona 3.x to połączenie jest kompletne: klasy są teraz typami i typy klasami. Dzisiaj klasy wciąż odzwierciedlają zarówno połączenie tych dwóch pojęć, jak i ich implementację.

Wraz z tymi zmianami Python rozwinął się jako spójny protokół kodowania *metaklas*, które są klasami potomnymi klasy `type`, przechwytyując wywołania tworzące klasy i zawierające kod przejmowany przez te klasy. W tym celu posiadają mechanizmy zarządzania i modyfikacji obiektów klas. Są zaawansowanym zagadnieniem, które dla większości programistów Pythona stanowi zagadnienie opcjonalne, zatem na razie nie będziemy się zagłębiać w jego szczegóły. Do metaklas wrócimy na chwilę w dalszej części rozdziału przy okazji omawiania dekoratorów klas, a szczegółowo zajmiemy się nimi w rozdziale 40., na zakończenie książki. Tutaj na razie zajmijmy się kilkoma innymi rozszerzeniami dotyczącymi klas.

Metody statyczne oraz metody klasy

W wersjach Pythona starszych od 2.2 funkcji metod klas nie można było nigdy wywoływać bez instancji — *metody statyczne* działają podobnie jak zdefiniowane w klasie funkcje nieprzyjmujące instancji, natomiast *metody klas* przyjmują obiekt klasy zamiast obiektu instancji. Oba rodzaje metod są podobne do spotykanych w innych językach (np. metod

statycznych w C++). Choć zostały wprowadzone wraz z klasami w nowym stylu, działają również w klasycznym modelu klas.

Aby aktywować te specjalne tryby działania metod, należy w klasie wywołać dedykowane funkcje wbudowane `staticmethod` i `classmethod` lub wykorzystać składnię `@nazwa` dekoratora, o czym powiemy więcej w dalszej części rozdziału. Funkcje te są niezbędne do włączenia specjalnego trybu w wersji 2.x i są ogólnie potrzebne w wersji 3.x. W Pythonie 3.0 metody wywoływane z klasy bez podawania instancji nie wymagają deklaracji `staticmethod`, ale takie metody wywoływanie z instancji już takiej deklaracji wymagają.

Do czego potrzebujemy metod specjalnych

Jak mieliśmy okazję się dowiedzieć, metoda klasy z reguły w swoim pierwszym argumencie wymaga podania instancji, która jest wykorzystywana jako podmiot wykonywanej metody — jest to „obiekt” w „programowaniu obiektowym”. Istnieją jednak dwa sposoby modyfikacji tego domyślnego działania. Zanim wytłumaczę, na czym polegają, należy wyjaśnić, do czego to może być przydatne.

Czasem programiści potrzebują przetwarzanie dane związane z klasami, nie z instancjami. Weźmy na przykład śledzenie liczby instancji utworzonych z klasy lub zarządzanie listą instancji klasy istniejących aktualnie w pamięci. Informacje tego typu i ich przetwarzanie to zadanie dla klasy, nie jej instancji. A dokładniej: informacje tego typu są najczęściej zapisywane w obiekcie klasy i przetwarzane bez użycia instancji.

Do zadań tego typu najczęściej wystarczą zwykłe funkcje zakodowane poza klasą: funkcje mają dostęp do atrybutów klasy za pośrednictwem jej nazwy, dzięki czemu mają też dostęp do danych zapisanych bezpośrednio w klasie i nie potrzebują tworzyć instancji. Jednak lepiej jest powiązać kod tego typu z klasą, której dotyczy, pozwalając na powtórne jego użycie przy wykorzystaniu dziedziczenia. Kod ten najlepiej jest zaimplementować w postaci metod klasy. Aby jednak takie metody zadziałyły zgodnie z oczekiwaniemi, nie powinny wymagać przekazywania instancji w argumencie `self`.

Python implementuje takie mechanizmy w formie tak zwanych *metod statycznych* — zwykłych funkcji nieobsługujących argumentu `self`, zdefiniowanych w ciele klasy i zaprojektowanych z myślą o współpracy z obiektem klasy, a nie — jak zwykłe metody — z obiektem instancji. Metodom statycznym nie jest automatycznie przekazywany argument `self`, niezależnie od tego, czy są wywoływane za pomocą klasy, czy instancji. Dane zapisane w klasie są dostępne z wszystkich instancji i nie są powiązane z żadną z nich.

Python obsługuje również metody klas, choć technika ta jest rzadziej stosowana. *Metody klas* to takie metody, którym zamiast instancji przekazywana jest klasa, niezależnie od tego, czy zostały wywołane z klasy, czy z instancji. Metody tego typu mają dostęp do danych klasy za pośrednictwem klasy przekazanej w atrybucie — wspomnianym wcześniej `self`, nawet w przypadku wywołania za pomocą instancji. Zwykłe metody (nazywane obecnie *metodami instancji*) otrzymują w pierwszym atrybucie instancję, z której zostały wywołane. W przypadku metod statycznych i metod klas tak się nie dzieje.

Metody statyczne w 2.x i 3.x

Koncepcja metod statycznych nie różni się w Pythonie 2.x i 3.x, ale wymagania w stosunku do jej implementacji zmieniły się nieco w Pythonie 3.x. Niniejsza książka omawia obie wersje Pythona, zatem należy wyjaśnić różnice w obydwu modelach, zanim przejdziemy do przykładowego kodu.

W rzeczywistości omawianie tego zagadnienia rozpoczęliśmy już w poprzednim rozdziale, przy okazji metod niezwiązanych. Jak pamiętamy, Python 2.x i 3.x metodom wywoływanym z

instancji zawsze przekazują tę instancję. Jednak w Pythonie 3.x metody wywoływane z klas są traktowane inaczej niż w 2.x i nie ma to związku z klasami w nowym stylu:

- w wersjach 2.x i 3.x tworzona jest *metoda związana*, jeżeli jest pobierana przez instancję;
- w Pythonie 2.x metoda w kontekście klasy jest traktowana jak *metoda niezwiązana*, której nie można wywołać, nie przekazując jej instancji w sposób jawny;
- w Pythonie 3.x metoda w kontekście klasy jest traktowana jak *zwyczka funkcja*, którą można wywołać bez podawania instancji.

Innymi słowy, metody klas w Pythonie 2.x wymagają przekazania im instancji, niezależnie od tego, czy są wywoływane z instancji, czy z klasy. Natomiast w Pythonie 3.x instancję musimy przekazać metodzie jedynie wówczas, gdy jej wymaga — metody, które nie mają zadeklarowanego argumentu `self`, mogą być wywoływane z klasy bez przekazywania instancji. Oznacza to, że wersja 3.x pozwala na wykorzystanie w klasie zwykłych funkcji, pod warunkiem, że nie oczekują argumentu instancji i że im takiego argumentu nie będziemy podawać. W efekcie mamy następującą sytuację:

- w Pythonie 2.x metodę zawsze musimy deklarować jako statyczną, jeśli chcemy ją wykorzystać bez instancji, niezależnie, czy jest wywoływana z klasy, czy z instancji;
- w Pythonie 3.x metod nie musimy deklarować jako statyczne, jeśli są wywoływane wyłącznie z klas, ale musimy to zrobić, jeśli chcemy je wywoływać z instancji.

Załóżmy na przykład, że chcemy wykorzystać atrybuty klasy do zliczania tego, jak dużo instancji zostało wygenerowanych z klasy. Poniższy kod (plik `spam.py`) przedstawia pierwsze podejście. Klasa zawiera licznik w postaci atrybutu, konstruktora zwiększającego ten licznik o jeden w chwili utworzenia instancji oraz metodę wyświetlającą wartość licznika. Należy pamiętać, że atrybuty klasy są współdzielone przez wszystkie instancje. Z tego powodu zapisanie atrybutu w samej klasie daje pewność, że jego wartość będzie dostępna we wszystkich instancjach.

```
class Spam:  
    numInstances = 0  
  
    def __init__(self):  
        Spam.numInstances = Spam.numInstances + 1  
  
    def printNumInstances():  
        print("Liczba utworzonych instancji: ", Spam.numInstances)
```

Metoda `printNumInstances` została napisana w taki sposób, aby dane przechowywała w klasie, nie w instancji, ponieważ wynik dotyczy *wszystkich* instancji. Z tego powodu chcemy mieć możliwość uruchamiania jej bez przekazywania instancji. Co więcej, nie chcemy tworzyć nowej instancji do odczytu licznika, ponieważ to spowodowałoby zmianę wartości tego licznika. Innymi słowy, potrzebujemy metody statycznej niewykorzystującej obiektu `self`.

Prezentowany kod jest prawidłowy w niektórych wersjach Pythona, ale nie działa w innych; znaczenie ma również sposób użycia tej klasy, czyli to, czy metoda `printNumInstances` jest wywoływana z klasy, czy z instancji. W Pythonie 2.x wywołania metod nieposiadających argumentu `self` nie będą działały, niezależnie od tego, czy wywołamy je z instancji, czy z klasy (część komunikatu o błędzie została pominięta).

```
C:\code> c:\python27\python  
>>> from spam import Spam  
  
>>> a = Spam() # W 2.x metod niezwiązań nie  
można wywoływać z klasy
```

```

>>> b = Spam()                                # Metody oczekują obiektu self
>>> c = Spam()
>>> Spam.printNumInstances()

TypeError: unbound method printNumInstances() must be called with Spam
instance as first argument (got nothing instead)

>>> a.printNumInstances()

TypeError: printNumInstances() takes no arguments (1 given)

```

Problemem jest to, że metody klas bez wiązania nie są w 2.x do końca tym samym co proste funkcje. Mimo że w deklaracji `def` nie ma argumentów, metoda nadal oczekuje przekazania instancji, ponieważ funkcja jest związaną z klasą. W Pythonie 3.x wywołanie metod bez `self` zadziała w przypadku wywołania z klasy, ale w przypadku wywołania z instancji zakończy się błędem.

```

C:\code> c:\python33\python
>>> from spam import Spam
>>> a = Spam()                                # W 3.x można wywoływać metody z
                                               klas
>>> b = Spam()                                # Jednak wywołania metod z
                                               instancji przekazują self
>>> c = Spam()
>>> Spam.printNumInstances()                  # Różnica w 3.x
                                              
Liczba utworzonych instancji: 3
>>> a.printNumInstances()

TypeError: printNumInstances() takes 0 positional arguments but 1 was given

```

Wywołanie metod nieobsługujących `self`, jak `printNumInstances`, nie zadziała w Pythonie 2.x, ale w 3.x już tak. Z drugiej strony, tego typu wywołania z instancji nie będą działały niezależnie od wersji Pythona, ponieważ *instancja* jest automatycznie przekazywana do metody, nawet jeśli ta metoda nie posiada argumentu, w którym mogłyby ją przyjąć.

```

Spam.printNumInstances()                      # Nie działa w 2.x, działa w 3.x
                                              
instance.printNumInstances()                  # Nie działa w 2.x i 3.x (chyba
                                               że jest to metoda statyczna)

```

Jeśli ktoś zdecyduje się na wykorzystanie Pythona 3.x i metody nieobsługujące `self` będzie wywoływały wyłącznie z klasy, to w zasadzie ma gotowy mechanizm metod statycznych. Jeśli jednak zdecyduje się na wykorzystanie takich metod w 2.x lub potrzebuje wywoływać je z instancji w 2.x i 3.x, będzie potrzebował dodatkowego mechanizmu adaptującego metody do takiego zastosowania albo powinien oznaczyć je jako specjalne. Przyjrzyjmy się każdemu z tych sposobów.

Alternatywy dla metod statycznych

Oprócz zadeklarowania metody jako specjalnej (nieprzymająccej argumentu `self`) programista ma do dyspozycji jeszcze kilka narzędzi pozwalających uzyskać zbliżony efekt. Jeśli potrzebujemy funkcji, które będą wykonywać działania na obiektach klas bez przekazywania instancji, najprostszym rozwiązaniem jest po prostu wykorzystanie zwykłych funkcji

zakodowanych niezależnie od definicji klasy. Dzięki temu w wywołaniu nie jest automatycznie przekazywana instancja. Poniższy przykład, będący modyfikacją modułu *spam.py*, działa bez przeszkód w 3.x i 2.x:

```
def printNumInstances():
    print("Liczba utworzonych instancji: ", Spam.numInstances)

class Spam:
    numInstances = 0

    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

C:\code> c:\python33\python
>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()          # Funkcja jest oderwana od klasy i nie
                                    # można
Liczba utworzonych instancji: 3      # nią manipulować w ramach
dziedziczenia
>>> spam.Spam.numInstances
3
```

Nazwa klasy jest dostępna dla zwykłej funkcji jako nazwa globalna, zatem kod zadziała bez przeszkód. Warto zwrócić uwagę, że nazwa funkcji staje się nazwą globalną, ale tylko w ramach modułu, w którym jest zdefiniowana, zatem nie będzie kolidować z nazwami zdefiniowanymi w innych plikach programu.

Zanim w Pythonie pojawiły się metody statyczne, właśnie taki był domyślny sposób realizacji tego typu zadań. Python oferuje moduły jako mechanizm kontroli przestrzeni nazw, zatem można by dyskutować, czy w ogóle jest sens umieszczać metody klas w definicji klas. Proste funkcje zaimplementowane w prezentowany sposób będą realizowały te same zadania co metody klas, a będą związane z klasami dzięki umieszczeniu ich definicji w tym samym module.

Niestety, takie podejście ma jednak sporo wad. Po pierwsze, w przestrzeni nazw modułu pojawia się osobna nazwa używana do przetwarzania tylko jednej klasy. Po drugie, funkcja jest tylko pośrednio powiązana z klasą, a jej definicja może znajdować się w zupełnie innym miejscu kodu. Co gorsza, proste funkcje nie mogą być przekazywane i przesyłane przez dziedziczenie, ponieważ istnieją poza przestrzenią nazw klasy, zatem klasy potomne nie będą brane pod uwagę przez funkcję i jedynym rozwiązaniem tego problemu jest ponowna definicja funkcji dla każdej klasy potomnej.

Można spróbować rozszerzyć ten kod w sposób bardziej przenośny, wykorzystując zwykłą metodę instancji i dbając o to, aby zawsze wywoływać ją z instancji.

```
class Spam:
    numInstances = 0

    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
```

```

def printNumInstances(self):
    print("Liczba utworzonych instancji: ", Spam.numInstances)

C:\code> c:\python33\python
>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created: 3
>>> Spam.printNumInstances(a)
Number of instances created: 3
>>> Spam().printNumInstances()           # odczyt licznika modyfikuje ten
licznik!
Liczba utworzonych instancji: 4

```

Niestety, jak wspomniałem wcześniej, tego typu podejście nie sprawdza się, jeśli nie mamy pod ręką gotowej instancji, ponieważ utworzenie nowej spowoduje zmianę licznika instancji zapisanego w klasie, co demonstruje ostatnie wywołanie przykładu. Lepsze rozwiązanie polega na oznaczeniu metody w klasie jako specjalnego przypadku, nieoczekującego obiektu instancji. Szczegóły poniżej.

Używanie metod statycznych i metod klas

W najnowszych wersjach Pythona istnieje jeszcze jedna opcja pozwalająca tworzyć proste funkcje skojarzone z klasami, które mogą być wywoływane z klasy lub instancji. Od Pythona 2.2 możemy tworzyć klasy wyposażone w metody statyczne i metody klas. Żadna z tych specjalnych odmian metod nie wymaga przekazania instancji. Metody tych typów muszą być oznaczone za pomocą wywołania funkcji wbudowanej `staticmethod` lub `classmethod` (wspominałem o tym przy okazji omawiania klas w nowym stylu). Obie z tych funkcji oznaczają metodę jako specjalną, to znaczy niewymagającą przekazania instancji w przypadku metod statycznych oraz wymagającą przekazania klasy w przypadku metod klas. Poniżej przedstawiony jest przykład (zawarty w pliku `bothmethods.py`, w którym wyświetlanie wyników zostało ujednolicone dla wersji 2.x i 3.x za pomocą list, choć w przypadku zwykłych klas w wersji 2.x wynik może być nieco inny):

```

# Plik bothmethods.py

class Methods:

    def imeth(self, x):                  # Zwykła metoda instancji: otrzymuje self
        print([self, x])

    def smeth(x):                      # Metoda statyczna: instancja nie jest
przekazywana
        print([x])

    def cmeth(cls, x):                 # Metoda klasy: otrzymuje klasę, nie
instancję
        print([cls, x])

    smeth = staticmethod(smeth)       # Przekształcenie smeth w metodę statyczną
(lub @ - patrz dalej)

```

```
cmeth = classmethod(cmeth)      # Przekształcenie cmeth w metodę klasy
(lub @ - patrz dalej)
```

Warto zwrócić uwagę, w jaki sposób dwa ostatnie wyrażenia *zmieniają przypisania* metod smeth i cmeth. Atrybuty są tworzone i modyfikowane przez przypisania w ciele klasy, zatem tego typu przypisania są w pełni standardowe i pozwalają zmienić pierwotną definicję metod. Jak się za chwilę przekonamy, można tu zastosować specjalną składnię ze znakiem @, tak jak w przypadku właściwości. Nie ma to jednak większego sensu, dopóki nie pozna się automatyzowanego w ten sposób procesu przypisania.

Z technicznego punktu widzenia Python pozwala na stosowanie trzech typów metod:

- *metod instancji*, którym przekazywany jest obiekt instancji `self` (domyślnie);
- *metod statycznych*, którym nie jest przekazywany żaden dodatkowy obiekt (za pomocą `staticmethod`);
- *metod klas*, którym przekazywany jest obiekt klasy (za pomocą `classmethod` i standardowo w metaklasach).

Dodatkowo w Pythonie 3.x umożliwiono stosowanie zwykłych funkcji w ciele klasy, które pełnią rolę metod statycznych w przypadku wywołania z klasy. Pomimo swej nazwy moduł `bothmethods.py` ilustruje wszystkie trzy rodzaje metod, zatem przyjrzymy się teraz każdej z nich.

Metody instancji to domyślna, najczęściej stosowana (również w tej książce) forma metod. Metoda instancji musi być wywołana za pomocą obiektu instancji. Jeśli metoda instancji jest wywołana z instancji, Python automatycznie przekaże do niej właśnie tę instancję w pierwszym argumentie, a jeśli zostanie wywołana z klasy, programista musi ręcznie przekazaćinstancję:

```
>>> from bothmethods import Methods      # Zwykłe metody instancji
>>> obj = Methods()                      # Utworzenie instancji
>>> obj.imeth(1)
[<bothmethods.Methods object at 0x0000000002A15710>, 1]
>>> Methods.imeth(obj, 2)
[<bothmethods.Methods object at 0x0000000002A15710>, 2]
```

Metody statyczne wywołuje się bez przekazywania obiektu instancji. W przeciwieństwie do zwykłych funkcji zdefiniowanych poza klasą nazwy metod statycznych są lokalne w przestrzeni nazw klasy i mogą być wykorzystywane w klasach potomnych. Metody, w których nie zdefiniowano argumentu instancji, mogą w 3.x być wywoływane w zwykły sposób, co nie jest możliwe w wersji 2.x. W 3.x `staticmethod` pozwala wywoływać metody statyczne również z instancji, a w wersji 2.x zarówno z klasy, jak i z instancji (tj. pierwszy poniższy sposób bez użycia deklaracji `staticmethod` działa w wersji 3.x poprawnie, ale drugi już nie):

```
>>> Methods.smeth(3)                  # Metoda statyczna, wywoływana z klasy
3                                     # Instancja nie jest przekazywana ani
oczekiwana
>>> obj.smeth(4)                    # Metoda statyczna, wywoływana z instancji
4                                     # Instancja nie jest przekazywana
```

Metody klas działają podobnie do zwykłych metod, ale Python zamiast instancji przekazuje do nich klasę w pierwszym argumencie, niezależnie od tego, czy zostały wywołane z klasy, czy z instancji.

```
>>> Methods.cmeth(5)                # Metoda klasy, wywoływana z klasy
```

```

<class '__main__.Methods'> 5      # Wywoływana jako cmeth(Methods, 5)
>>> obj.cmeth(6)                  # Metoda klasy, wywoływana z instancji
<class '__main__.Methods'> 6      # Wywoływana jako cmeth(Methods, 6)

```

W rozdziale 40. pokażę, że *metody metaklasy* (unikatowy, zaawansowany i technicznie odmienny rodzaj metod) działają podobnie do opisanych tutaj jawnie zadeklarowanych metod klasy.

Zliczanie instancji z użyciem metod statycznych

Wyposażeni w funkcję wbudowaną `staticmethod` możemy pokusić się o ponowną implementację klasy zliczającej swoje instancje. Funkcja `staticmethod` oznacza metodę jako specjalną, dlatego Python nie przekaże jej obiektu instancji.

```

class Spam:
    numInstances = 0                      # Użycie metod statycznych dla
    danych klasy

    def __init__(self):
        Spam.numInstances += 1

    def printNumInstances():
        print("Liczba instancji: %s" % Spam.numInstances)

    printNumInstances = staticmethod(printNumInstances)

```

Dzięki zadeklarowaniu metody jako statycznej można ją wywołać z klasy lub z dowolnej instancji i uzyskamy dokładnie taki sam efekt, zarówno w Pythonie 2.x, jak i 3.x.

```

>>> from spam_static import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()           # Wywoływana jak zwykła funkcja
Liczba instancji: 3
>>> a.printNumInstances()            # Instancja nie jest
przekazywana
Liczba instancji: 3

```

Porównując to z wydzieleniem metody `printNumInstances` poza ciało klasy (co uczyniliśmy wcześniej), użyliśmy specjalnej funkcji `staticmethod` (można też użyć znaku @, o czym za chwilę). Dzięki temu nazwa metody pozostała lokalna w klasie (nie będzie powodować konfliktów z innymi nazwami w module), kod funkcji znajduje się bliżej miejsca jego użycia (wewnątrz ciała klasy), a klasy podzielone mogą dostosowywać metodę statyczną, wykorzystując wszelkie zalety dziedziczenia. Jest to wygodniejsze i lepsze rozwiązanie niż importowanie funkcji z pliku zawierającego zakodowaną klasę nadzczną. Ilustruje to poniższa klasa podzielona i listing nowej sesji testowej (należy pamiętać o otwarciu nowej sesji po zmodyfikowaniu pliku, aby instrukcja `from` zimportowała najnowszą wersję pliku):

```
class Sub(Spam):
```

```

def printNumInstances():                      # Przeciążenie metody statycznej
    print("Coś ekstra...")
    Spam.printNumInstances()                  # Ale z wywołaniem oryginału

Spam.printNumInstances()

printNumInstances = staticmethod(printNumInstances)

>>> from spam_static import Spam, Sub
>>> a = Sub()
>>> b = Sub()
>>> a.printNumInstances()                   # Wywołanie z instancji klasy
potomnej
Coś ekstra...
Liczba instancji: 2
>>> Sub.printNumInstances()                # Wywołanie z samej klasy
potomnej
Coś ekstra...
Liczba instancji: 2
>>> Spam.printNumInstances()              # Wywołanie oryginalnej wersji
Liczba instancji: 2

```

Klasy potomne mogą dziedziczyć metodę statyczną bez jej ponownego definiowania. Metodę taką wywołuje się bez przekazywania instancji, niezależnie od tego, w którym miejscu drzewa dziedziczenia została zdefiniowana.

```

>>> class Other(Spam): pass             # Dziedziczenie metody
statycznej
>>> c = Other()
>>> c.printNumInstances()
Liczba instancji: 3

```

Warto zwrócić uwagę, że w powyższym rozwiążaniu zwiększa się licznik instancji *klasy nadzędnej*, ponieważ konstruktor jest dziedziczony i uruchamiany — efekt ten jest tematem następnego podrozdziału.

Zliczanie instancji z metodami klas

Co interesujące, tę samą funkcję zliczania instancji można zaimplementować z użyciem *metod klas*. Poniższy listing daje dokładnie takie same wyniki jak wersja z użyciem metody statycznej, z tą różnicą, że metoda klasy otrzymuje w swoim pierwszym argumentem obiekt klasy, z której została wywołana. Dzięki temu w kodzie klasy nie trzeba kodować nazwy klasy, co czyni ją bardziej modularną.

```

class Spam:
    numInstances = 0                      # Użycie metody klasy
    def __init__(self):
        Spam.numInstances += 1

```

```

def printNumInstances(cls):
    print("Liczba instancji: %s" % cls.numInstances)
printNumInstances = classmethod(printNumInstances)

```

Ta klasa jest używana w taki sam sposób jak poprzednie wersje, ale jej metoda `printNumInstances`, niezależnie od tego, czy została wywołana z klasy, czy z instancji, otrzymuje w pierwszym argumencie obiekt klasy.

```

>>> from spam_class import Spam
>>> a, b = Spam(), Spam()
>>> a.printNumInstances()                               # Klasa przekazana w pierwszym
argumencie
Liczba instancji: 2
>>> Spam.printNumInstances()                         # Klasa przekazana w pierwszym
argumencie
Liczba instancji: 2

```

W przypadku używania metod klas należy zapamiętać, że zawsze przekazywana jest im klasa znajdująca się na *najniższym* poziomie drzewa dziedziczenia. Ta właściwość może mieć pewne subtelne konsekwencje w przypadku prób modyfikacji danych przekazanej klasy. Weźmy na przykład moduł `spam_class.py`, który zmodyfikujemy w taki sposób, aby metoda `Spam.printNumInstances` wyświetlała również wartość argumentu `cls`.

```

class Spam:
    numInstances = 0                                # Śledzenie liczby instancji
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Liczba instancji: %s %s" % (cls.numInstances, cls))
    printNumInstances = classmethod(printNumInstances)
class Sub(Spam):
    def printNumInstances(cls):                      # Przeciążenie metody klasy
        print("Coś ekstra...", cls)                 # Ale z wywołaniem oryginału
        Spam.printNumInstances()
    printNumInstances = classmethod(printNumInstances)
class Other(Spam): pass                           # Dziedziczenie metody klasy

```

Przy wywołaniu metody klasy przekazywana jest klasa najbliższego poziomu, nawet w przypadku klas potomnych, które nie mają zdefiniowanych metod klasy.

```

>>> from spam_class import Spam, Sub, Other
>>> x = Sub()
>>> y = Spam()

```

```

>>> x.printNumInstances()                                # Wywołanie z instancji klasy
potomnej

Coś ekstra... <class 'test.Sub'>

Liczba instancji: 2 <class 'spam_class.Spam'>

>>> Sub.printNumInstances()                           # Wywołanie z samej klasy
potomnej

Coś ekstra... <class 'test.Sub'>

Liczba instancji: 2 <class 'spam_class.Spam'>

>>> y.printNumInstances()                            # Wywołanie z instancji klasy
nadrzędnej

Liczba instancji: 2 <class 'spam_class.Spam'>

```

W pierwszym wywołaniu metoda klasy jest wywoływana z instancji klasy Sub, Python przekazuje do niej obiekt klasy Sub. W tym przypadku wszystko działa, jak należy, ponieważ metoda w klasie Sub wywołuje metodę klasy nadrzędnej (Spam), dzięki czemu metoda o tej samej nazwie w Spam otrzyma w pierwszym argumencie obiekt klasy Spam. Sprawdzmy jednak, co się stanie, gdy metodę klasy wywołamy z instancji klasy potomnej:

```

>>> z = Other()                                     # Wywołanie z niższej
instancji klasy potomnej

>>> z.printNumInstances()

Liczba instancji: 3 <class 'spam_class.Other'>

```

Ostatnie wywołanie przekaże obiekt klasy Other do metody klasy Spam. W naszym przypadku nadal wynik będzie zgodny z oczekiwaniem, ponieważ *odczytywany* licznik jest zdefiniowany w Spam i zostanie odnaleziony w tej klasie. Gdyby jednak metoda usiłowała *zapisać* dane w klasie przekazanej jej w pierwszym argumencie, zapis zostałby wykonany do klasy Other, nie Spam. W tym szczególnym przypadku zamiast zdawać się na przekazywany klasie argument, prawdopodobnie lepiej będzie wpisać nazwę własnej klasy, jeżeli mają być zliczane również instancje wszystkich klas podrzędnych.

Zliczanie instancji dla każdej z klas z użyciem metod klas

Z faktu, że metody klas otrzymują obiekt klasy z najbliższego poziomu w drzewie dziedziczenia wynikają następujące właściwości:

- metody *statyczne* i wykorzystanie jawnych nazw klas są lepszą strategią zapisywania danych wspólnych dla wszystkich klas z drzewa.
- metody *klas* są lepszą strategią jeśli dane mają być gromadzone dla każdej klasy indywidualnie.

Przykładem problemu, do którego lepiej nadają się metody klas może być zliczanie instancji *każdej klasy* z drzewa dziedziczenia. W poniższym listingu w klasie najwyższego poziomu zdefiniowano metodę klasy zapisującą informację stanu, która ma być zapisana dla każdej klasy niezależnie. Jest to zachowanie analogiczne do działania metod instancji, które zapisują dane niezależnie w każdej instancji.

```

class Spam:

    numInstances = 0

    def count(cls):                               # Licznik instancji zapisany w klasie

```

```

        cls.numInstances += 1           # cls jest obiektem klasy najniższego
poziomu

    def __init__(self):
        self.count()                 # Przekazuje self.__class__

    count = classmethod(count)

class Sub(Spam):
    numInstances = 0

    def __init__(self):           # Przeciążenie metody __init__
        Spam.__init__(self)

class Other(Spam):             # Odziedziczenie metody __init__
    numInstances = 0

>>> from spam_class2 import Spam, Sub, Other
>>> x = Spam()
>>> y1, y2 = Sub(), Sub()
>>> z1, z2, z3 = Other(), Other(), Other()
>>> x.numInstances, y1.numInstances, z1.numInstances      # Niezależne dane
w każdej klasie!
(1, 2, 3)
>>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)

```

Metody statyczne i metody klasy mają dodatkowe zaawansowane role, których nie będziemy omawiać w tym miejscu. Więcej przykładów jest zawartych w innych materiałach. W najnowszych wersjach Pythona deklarowanie metod statycznych i metod klas stało się jeszcze prostsze dzięki wprowadzeniu *dekoratorów funkcji*. Jest to prosty sposób wykonania funkcji na innej funkcji, którego możliwe zastosowania znacznie wykraczają poza deklarację metod statycznych i metod klas. Dekoratory mogą również być w Pythonie 2.x i 3.x wykorzystywane do modyfikowania zachowania klas, na przykład do inicjalizacji licznika numInstances. Przykład takiego użycia przedstawia następny punkt rozdziału.



Kończąc opis typów metod w Pythonie: warto zapoznać się z opisem *metod metaklas* w rozdziale 40. Ponieważ metody te służą do przetwarzania klas będących instancjami metaklas, są bardzo podobne do zdefiniowanych tu metod klas. Nie wymagają jednak deklaracji classmethod i dotyczą jedynie mrocznej domeny metaklas.

Dekoratory i metaklasy – część 1.

Ponieważ opisana w poprzednim podrozdziale technika z wywołaniem funkcji staticmethod i classmethod wydawała się niektórym użytkownikom zbyt toporna, dodano nową opcję, która może ją uprościć. *Dekoratory funkcji* (ang. *function decorator*) — podobne pod względem przeznaczenia i składni do adnotacji w języku Java — zarówno spełniają tę specyficzną

potrzebę, jak również stanowią podstawowe narzędzie do dodawania kodu zarządzającego funkcjami, klasami i ich późniejszymi wywołaniami.

Powysza funkcjonalność nosi nazwę „dekorowania”. W rzeczywistości jest to sposób wykonywania za pomocą specjalnej składni dodatkowych kroków przetwarzających funkcję lub klasę na etapie jej definiowania. Są dwie odmiany dekoratorów:

- *Dekoratory funkcji*: pierwsza odmiana wprowadzona w wersji 2.4, rozszerzająca definicję funkcji. Definiuje specjalny tryb operacji zarówno dla prostych funkcji, jak i metod klas poprzez opakowywanie ich w dodatkową warstwę logiki zaimplementowanej w postaci innej funkcji, zwanej *metafunkcją*.
- *Dekoratory klas*: późniejsze rozszerzenie wprowadzone w wersjach 2.6 i 3.0, zmieniające definicję klasy. Pozwala zarządzać całymi obiektami i ichinstancjami. Jest nieco prostsze od metaklasy, ale często pełni podobną rolę.

Dekoratory funkcji okazują się narzędziami uniwersalnymi — przydają się do dodawania do funkcji dodatkowej logiki, możliwości wykraczają znacznie poza deklarację metody jako statycznej. Można je na przykład wykorzystać do rozszerzenia funkcji za pomocą kodu logującego ich wywołania czy sprawdzającego typy argumentów przekazanych w czasie debugowania. Dekoratory funkcji mogą zarządzać samymi funkcjami lub ich późniejszymi wywołaniami. Pod tym drugim względem dekoratory funkcji są podobne do opisanego w rozdziale 31. wzorca projektowego *delegacji*, jednak zaprojektowano je z myślą o rozszerzeniu określonego wywołania funkcji lub metody, a nie całego interfejsu obiektu.

Python udostępnia pewne wbudowane dekoratory dla operacji takich jak oznaczanie metod statycznych i metod klas czy definiowanie właściwości (jak wspomniałem wcześniej, wbudowana funkcja `property` automatycznie pełni funkcję dekoratora), jednak programiści mogą również samodzielnie tworzyć dowolne dekoratory. Choć nie są one ściśle powiązane z klasami, dekoratory funkcji zdefiniowane przez użytkownika często zapisywane są jako klasy w celu zachowania oryginalnych funkcji wraz z innymi danymi, takimi jak informacje o stanie.

Dekoratory okazały się tak przydatną funkcjonalnością, że zostały rozszerzone w wersjach Pythona 2.6, 2.7 i 3.x. Rozbudowują również klasy i są ściślej związane z ich modelem. Podobnie jak ich funkcyjne odpowiedniki mogą zarządzać samymi klasami, jak również tworzonymi później instancjami. W tym drugim trybie często wykorzystują *delegacje*. Jak się przekonamy, rola dekoratorów klas częściowo pokrywa się z tym, do czego służą *metaklasy*. Nowe dekoratory klas często oferują lepsze sposoby osiągania tych samych celów.

Podstawowe informacje o dekoratorach funkcji

Z punktu widzenia składni dekorator funkcji jest rodzajem deklaracji w czasie wykonywania dotyczącej funkcji, która następuje po nim. Dekorator funkcji zapisywany jest w wierszu tuż przed instrukcją `def` definiującą funkcję lub metodę i składa się z symbolu `@`, po którym następuje coś, co nazywamy *metafunkcją* — funkcja (lub inny obiekt wywoływalny), która zarządza inną funkcją. Począwszy od wersji Pythona 2.4, metody statyczne można na przykład tworzyć za pomocą poniższej składni dekoratora.

```
class C:  
    @staticmethod  
        # Składnia dekoratora  
    def meth():  
        ...
```

Wewnętrznie składnia ta ma ten sam efekt co poniższa (przekazanie funkcji przez dekorator oraz przypisanie wyniku z powrotem do oryginalnej nazwy).

```
class C:
```

```
def meth():
    ...
meth = staticmethod(meth) # Ponowne wiązanie nazwy
```

Dekorator wiąże nazwę metody z wynikiem wywołania dekoratora. W efekcie wywołanie metody powoduje niejawne wcześniejsze wywołanie dekoratora. Dekorator może zwrócić dowolny obiekt, dzięki czemu dekoratory pozwalają przywiązać dodatkową logikę do każdego wywołania metody. Funkcja dekoratora może zwrócić oryginalną funkcję albo nowy obiekt pośredniczący, który ma zapisaną oryginalną funkcję, wywoływaną po uruchomieniu dodatkowej logiki implementowanej w dekoratorze.

Dzięki składni dekoratorów naszą implementację metod statycznych możemy zapisać w następujący sposób. Kod działa tak samo w Pythonie 2.x i 3.x.

```
class Spam:
    numInstances = 0

    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

    @staticmethod
    def printNumInstances():
        print("Liczba utworzonych instancji: ", Spam.numInstances)

>>> from spam_static_deco import Spam

>>> a = Spam()
>>> b = Spam()
>>> c = Spam()

>>> Spam.printNumInstances()          # Teraz działają wywołania z klas i z
instancji!
Liczba utworzonych instancji: 3
>>>a.printNumInstances()

Liczba utworzonych instancji: 3
```

Ponieważ argumentami i wynikami wbudowanych funkcji `classmethod` i `property` mogą być również funkcje, można ich używać w ten sam sposób jak dekoratorów. Ilustruje to poniższa odmiana kodu z opisanego wcześniej pliku `bothmethods.py`:

```

@classmethod
def cmeth(cls, x):          # Klasa: dostaje klasę, a nie instancję
    print([cls, x])

@property
def name(self):
    return 'Robert ' + self.__class__.__name__

>>> from bothmethods_decorators import Methods
>>> obj = Methods()
>>> obj.imeth(1)
[<bothmethods_decorators.Methods object at 0x00000000002A256A0>, 1]
>>> obj.smeth(2)
[2]
>>> obj.cmeth(3)
[<class 'bothmethods_decorators.Methods'>, 3]
>>> obj.name
'Robert Methods'

```

Należy pamiętać, że `staticmethod` jest funkcją wbudowaną. Można jej użyć ze składnią dekoratorów, ponieważ funkcja ta jako swój parametr przyjmuje inną funkcję i zwraca obiekt uruchamiany (ang. *callable*), z którym można powiązać nazwę oryginalnej funkcji. Każdej funkcji o takiej właściwości można użyć jako dekoratora, również funkcji napisanej przez użytkownika, co zademonstruje następny podrozdział.

Pierwsze spojrzenie na funkcję dekoratora zdefiniowaną przez użytkownika

Python oferuje spory wybór funkcji wbudowanych, których można używać w charakterze dekoratorów, możemy też pisać własne dekoratory. Z powodu ich wszechstronnego zastosowania dekoratorom poświęcimy cały rozdział w ostatniej części książki. W tym miejscu posłużymy się prostym przykładem dekoratora w działaniu.

Jak pamiętamy z rozdziału 30., metoda przeciążania operatorów `__call__` implementuje interfejs wywołania funkcji dla instancji klas. Poniższy kod wykorzystuje to w celu zdefiniowania klasy pośredniczącej zapisującą funkcję dekoratora w instancji i przechwytyującą wywołania do oryginalnej nazwy. Ponieważ jest to klasa, ma również informacje o stanie (licznik wykonanych wywołań).

```

class tracer:
    def __init__(self, func):      # Zapamiętanie oryginalnej funkcji,
        zainicjowanie licznika
        self.calls = 0
        self.func = func

    def __call__(self, *args):     # Przy późniejszych wywołaniach: dodanie
        kodu, uruchomienie oryginalnej funkcji

```

```

        self.calls += 1
        print('wywołanie %s do %s' % (self.calls, self.func.__name__))
    return self.func(*args)

@tracer                      # Równoważne wywołaniu spam = tracer(spam)
def spam(a, b, c):           # Opakowanie funkcji spam w obiekt dekoratora
    return a + b + c
print(spam(1, 2, 3))         # Naprawdę wywołuje obiekt opakowujący
tracer
print(spam('a', 'b', 'c'))   # Wywołuje __call__ w klasie

```

Ponieważ funkcja `spam` jest wykonywana przez dekorator `tracer`, kiedy wywołana zostaje oryginalna nazwa `spam`, tak naprawdę uruchomiona zostaje metoda `__call__` z klasy. Metoda ta odlicza i loguje wywołania, a następnie wywołuje oryginalną opakowaną funkcję. Warto zwrócić uwagę na wykorzystanie składni argumentów `*nazwa` w celu spakowania i rozpakowania przekazanych argumentów. Z tego powodu ten dekorator może zostać wykorzystany do opakowania dowolnej funkcji z dowolną liczbą argumentów.

Rezultat jest taki, że do oryginalnej funkcji `spam` dodana zostaje warstwa logiki. Poniżej widać dane wyjściowe tego skryptu w wersji 2.x i 3.x — pierwszy wiersz pochodzi z klasy `tracer`, drugi z funkcji `spam`.

```
c:\code> python tracer1.py
wywołanie 1 do spam
6
wywołanie 2 do spam
abc
```

By lepiej zrozumieć ten mechanizm, warto prześledzić kod tego przykładu. Prezentowany dekorator działa dla każdej funkcji przyjmującej parametry pozycyjne, ale nie zwraca wyników dekorowanej funkcji, nie obsługuje argumentów kluczowych i nie może być użyty do dekorowania metod klas (w skrócie: w przypadku metod do metody `__call__` dekoratora zostanie przekazana tylko instancja śledząca). Jak zobaczymy w części VIII, istnieje wiele sposobów tworzenia dekoratorów funkcji, między innymi zagnieżdżone instrukcje `def`. Niektóre z tych technik lepiej nadają się do obsługi metod niż prezentowana w powyższym przykładzie.

Na przykład dzięki wykorzystaniu zagnieżdżonych funkcji z otaczającym je zakresem widoczności stanu zamiast wywoływalnych instancji klas z atrybutami dekoratory funkcji są często stosowane z metodami na poziomie klas. Szczegółami zajmiemy się później, natomiast teraz przyjrzyjmy się ogólnie temu modelowi kodowania opartemu na *domknięciu*. Wykorzystuje on atrybuty funkcji dla stanu licznika, aby kod był uniwersalny. W wersji 3.x można jednak wykorzystać zmienne nielokalne.

```

def tracer(func):             # Zapamiętanie oryginalnej wartości
    def oncall(*args):          # dla późniejszych wywołań
        oncall.calls += 1
        print('wywołanie %s do %s' % (oncall.calls, func.__name__))
    return func(*args)
oncall.calls = 0

```

```

        return oncall

class C:
    @tracer
    def spam(self,a, b, c): return a + b + c

x = C()
print(x.spam(1, 2, 3))
print(x.spam('a', 'b', 'c'))      # Ten sam wynik jak funkcji tracer1 (w
tracer2.py)

```

Pierwsze spojrzenie na dekoratory klas i metaklasy

Dekoratory funkcji okazały się tak użyteczne, że w Pythonie 2.6 i 3.0 rozszerzono ich możliwości o współpracę z klasami. *Dekoratory klas* mają działanie zbliżone do dekoratorów funkcji, z tą różnicą, że są uruchamiane na końcu instrukcji `class`, wiążąc wynik swojego wykonania z nazwą klasy. Dzięki temu dekoratory klas mogą być wykorzystane do zarządzania klasami po ich utworzeniu, jak również do opakowywania logiki klas w dodatkową logikę, na przykład do zarządzania instancjami. Składnia użycia dekoratora klas jest następująca:

```

def decorator(aClass):
    ...
    @decorator          # Składnia dekoratora klasy
    class C:
        ...

```

Powyższy kod jest równoważny następującemu:

```

def decorator(aClass):
    ...
class C:          # Odpowiednik powiązania nazw
    ...
C = decorator(C)

```

Dekorator klas może modyfikować klasę, może też zwracać obiekt *pośredniczący*, który będzie przechwytywał wywołania konstruktora instancji. Za pomocą takiego dekoratora moglibyśmy w następujący sposób zmodyfikować zaprezentowaną w podrozdziale „Zliczanie instancji dla każdej z klas z użyciem metod klas” klasę zawierającą licznik instancji i inne wymagane dane:

```

def count(aClass):
    aClass.numInstances = 0
    return aClass          # Zwracamy samą klasę, nie obiekt
                           # opakowujący
@count
class Spam: ...          # Równoważne wywołaniu Spam = count(Spam)
@count

```

```

class Sub(Spam): ...           # numInstances = 0 nie jest potrzebne
@count
class Other(Spam): ...

```

W rzeczywistości zakodowany w ten sposób dekorator można stosować z klasami i funkcjami. Zwraca on obiekt zdefiniowany w obu kontekstach po zainicjowaniu atrybutu obiektu:

```

@count
def spam(): pass             # Podobnie jak spam = count(spam)
@count
class Other: pass            # Podobnie jak Other = count(Other)
spam.numInstances            # Oba ustawione na zero
Other.numInstances

```

Choć dekorator zarządza samą funkcją lub klasą, może również — jak się przekonamy w dalszej części książki — zarządzać całym *interfejsem* obiektu poprzez przechwytywanie wywołań konstrukcyjnych i opakowywanie nowego obiektu instancji w obiekt *pośredniczący*, który wdraża narzędzia dostępowe do atrybutów w celu przechwytywania późniejszych żądań. Jest to wielopoziomowa technika kodowania, którą wykorzystamy w rozdziale 39. do zaimplementowania prywatności atrybutu klasy. Tutaj pokazana jest zapowiedź tego modelu:

```

def decorator(cls):          # Podczas dekorowania @
    class Proxy:
        def __init__(self, *args):      # Podczas tworzenia instancji:
utworzenie cls
            self.wrapped = cls(*args)

        def __getattr__(self, name):    # Podczas pobierania atrybutu:
dodatkowe operacje
            return getattr(self.wrapped, name)

    return Proxy

@decorator
class C: ...      # Podobnie jak C = decorator(C)
X = C()          # Utworzenie obiektu pośredniczącego opakowującego klasę C i
przechwytyjącego późniejszy X.attr

```

Metaklasy są inną formą zaawansowanej mechaniki modyfikującej działanie klas. Ich funkcje przenikają się w pewnym zakresie z funkcjami dekoratorów klas. Metaklasa tworzy alternatywny model przejmujący procedurę konstrukcji obiektu instancji, przekazując ją do klasy potomnej klasy *type*.

```

class Meta(type):
    def __new__(meta, classname, supers, classdict):
        ...dodatkowy kod + tworzenie klasy za pomocą instrukcji type...
class C(metaclasse=Meta):
    ...tworzenie klasy przekierowane do Meta...      # Na przykład C = Meta('C',
(), {...})

```

W Pythonie 2.x efekt jest ten sam, ale stosuje się odmienną składnię: zamiast parametru słownikowego `metaclass` w deklaracji klasy należy w jej ciele zadeklarować atrybut klasy o nazwie `__metaclass__`.

```
class C:  
    __metaclass__ = Meta  
    ... tworzenie klasy przekierowane do Meta...
```

W obu wersjach Pythona wywoływana jest metaklasa w celu utworzenia nowego obiektu klasy z wykorzystaniem danych zdefiniowanych podczas wykonywania instrukcji `class`. W wersji 2.x metaklasa po prostu odpowiada klasycznemu kreatorowi klasy:

```
classname = Meta(classname, superclasses, attributedict)
```

Metaklasy najczęściej implementują dwie metody klasy `type`: `__new__` oraz `__init__`, przejmując kontrolę nad procedurą tworzenia i inicjalizacji nowej instancji klasy. W efekcie, podobnie jak ma to miejsce w przypadku dekoratorów klas, otrzymujemy nową logikę uruchamianą automatycznie na etapie tworzenia klasy. Ten krok wiąże nazwę klasy z wynikiem wywołania metaklasy zdefiniowanej przez użytkownika. W rzeczywistości metaklasa nie musi być w ogóle klasą — w tym przypadku (przeanalizujemy go później) zaciera się granica pomiędzy tym narzędziem a dekoratorami, a oba pojęcia pod wieloma względami można wręcz zakwalifikować jako równoważne.

Obydwa mechanizmy służą do modyfikowania działania klas, mogą też zwracać dowolne obiekty zastępujące klasy. Ten interfejs daje praktycznie nieograniczone możliwości w zakresie zarządzania klasami. Jak zobaczymy później, metaklasy mogą również definiować *metody* przetwarzające ich klasy instancji zamiast zwykłych instancji. Jest to technika podobna do metod klasy i można ją emulować za pomocą metod i danych w obiektach klasy dekoratora, a nawet dekoratora klasy zwracającego instancję metaklasy. Do zrozumienia takich zadziwiających pomysłów potrzebne będą podstawy koncepcyjne z rozdziału 40. (i prawdopodobnie również środki uspokajające!).

Dalsza lektura

Oczywiście dekoratory i metaklasy są zagadnieniami znacznie bardziej zaawansowanymi, niż udało mi się je tu przedstawić. Są pomyślane jako mechanizm bardzo ogólny, który może być wymagany w niektórych pakietach. Kodowanie nowych dekoratorów użytkownika i metaklas to zaawansowane zagadnienia, którymi zajmują się przede wszystkim twórcy narzędzi programistycznych, nie autorzy aplikacji. Z tego powodu szczegóły na ich temat odłożymy do ostatniej części książki:

- rozdział 38. demonstruje zasady tworzenia właściwości z wykorzystaniem składni dekoratora funkcji,
- rozdział 39. zawiera dalsze szczegóły dotyczące dekoratorów, w tym bardziej zaawansowane przykłady,
- rozdział 40. omawia metaklasy i pogłębia temat zarządzania klasami i instancjami.

Wspomniane wyżej rozdziały omawiają zaawansowane zagadnienia, ale zawierają też bardziej szczegółowe przykłady zastosowań Pythona, niż można przedstawić w pozostałej części książki. Teraz przejdźmy do ostatniego tematu dotyczącego klas.

Wbudowana funkcja super: zmiana na lepsze czy na gorsze?

Do tej pory wspominałem o wbudowanej funkcji `super` tylko mimo chodem, ponieważ jest ona stosowana stosunkowo rzadko, a jej użycie może wydawać się kontrowersyjne. Zważywszy jednak na jej rosnącą w ostatnich latach popularność, warto jej poświęcić w tym wydaniu nieco miejsca. Ten podrozdział nie tylko jest wprowadzeniem do funkcji `super`, ale stanowi również studium przypadku projektowania języka, zamykające rozdział poświęcony wielu różnym narzędziom, których istnienie może wydawać się dość osobliwe w języku skryptowym, jakim jest Python.

W niektórych miejscach tego podrozdziału kwestionowana jest różnorodność tych narzędzi, dlatego zachęcam do samodzielnego oceniania subiektywnych opinii (wróćmy do tych zagadnień na końcu książki po rozwinięciu tematu zaawansowanych narzędzi, takich jak metaklasy i deskryptory). Niemniej jednak szybki rozwój Pythona w ostatnich latach jest strategiczną decyzyją społeczności użytkowników, a funkcja `super` wydaje się jednym z wielu reprezentatywnych przykładów.

Wielka debata o funkcji `super`

Jak wspomniałem w rozdziałach 28. i 29., Python posiada wbudowaną funkcję `super`, która wykorzystuje się do generycznego wywoływania metod klasy nadzędnej. Opis tej funkcji został specjalnie odłożony aż do tego miejsca, ponieważ użyta w istniejącym kodzie ma istotne mankamenty i stosuje się ją w bardzo szczególnych przypadkach, które dla wielu użytkowników mogą być niejasne i skomplikowane. Dla większości początkujących programistów lepszy jest stosowany do tej pory tradycyjny schemat wywoływania funkcji jawnie za pomocą nazwy. Krótkie podsumowanie uzasadniające ten wybór zawarte jest w ramce „A co z funkcją `super`?” w rozdziale 28.

Społeczność użytkowników Pythona jest podzielona pod tym względem. W internecie można znaleźć całe spektrum artykułów na ten temat, począwszy od „Funkcja `super` w Pythonie jest szkodliwa” do „Funkcja `super` jest naprawdę super!”^[3]. Szczerze mówiąc, na moich kursach to wywołanie okazuje się najbardziej interesujące dla programistów Javy, którzy zaczynają używać Pythona. Przyczyną jest koncepcyjne podobieństwo tej funkcji do narzędzia w Javie (w końcu wiele nowych funkcji Pythona zawdzięcza swoje istnienie programistom innych języków, którzy wprowadzili swoje stare nawyki do nowego modelu). Jednak funkcja `super` w Pythonie nie jest tym samym, czym jest w Javie. Inaczej przekłada się w wielokrotnym dziedziczeniu i ma inne zastosowania. Niemniej jednak od czasu pojawienia się wywołuje kontrowersje i nieporozumienia.

Opis funkcji `super` został odłożony do tego miejsca (i niemal całkowicie pominięty w poprzednich wydaniach), ponieważ ma ona istotne wady. W wersji 2.x jest zbyt uciążliwa w użyciu, różni się formą w wersjach 2.x i 3.x, opiera się na nietypowej semantyce w wersji 3.x i źle wpasowuje się w wielokrotne dziedziczenie i przeciążanie operatorów w typowym kodzie. Jak się przekonamy, czasami może maskować problemy i zniechęcać do ścisłszego stylu kodowania dającego lepszą kontrolę.

Na obronę funkcji `super` należy przyznać, że w niektórych przypadkach jej użycie jest naprawdę uzasadnione. Dotyczy to kooperatywnego wywoływania metod o takich samych nazwach w diamentowym drzewie wielokrotnego dziedziczenia. Jest to jednak zbyt wymagający przypadek dla początkujących programistów. Funkcja `super` musi być stosowana w sposób uniwersalny i — można rzec — patologicznie spójny, podobnie jak opisany wcześniej atrybut `__slots__`. Opiera się na niezaprzeczalnie niejasnym algorytmie MRO i jest przeznaczona do przypadków, które zdecydowanie bardziej wydają się wyjątkami niż normami w Pythonie. Ze względu na swoją rolę jest to zaawansowane narzędzie oparte na osobliwych zasadach, które nie dotyczą większości użytkowników Pythona, a stosowanie go do osiągnięcia praktycznych celów wydaje się dość sztuczne. Oczekiwanie, że funkcja `super` znajdzie uniwersalne zastosowanie w ogromnej większości istniejącego kodu, wydaje się nierealne.

Z powyższych powodów w tej wprowadzającej książce dotychczas preferowany był tradycyjny, zalecany dla początkujących programistów schemat wywołań oparty na jawnych nazwach. Lepszym podejściem jest opanowanie najpierw tego tradycyjnego schematu i pozostanie przy nim niż stosowanie dodatkowych, wykorzystujących tajemną magię i przeznaczonych do przypadków specjalnych narzędzi, które w niektórych kontekstach mogą nie działać. Nie jest to wyłącznie opinia autora. Mimo usilnych starań obrońców funkcji `super` jej stosowanie nie jest obecnie uznawane za „dobrą praktykę” z całkowicie uzasadnionych powodów.

Z drugiej strony, podobnie jak w przypadku innych narzędzi, rosnąca w ostatnich latach popularność tej funkcji sprawia, że dla wielu programistów przestaje ona być opcjonalna. Dla kogoś, kto ją widzi po raz pierwszy, staje się wręcz oficjalnie obowiązkowa! Ten podrozdział jest przeznaczony dla czytelników, którzy chcą poeksperymentować z funkcją `super` lub muszą jej używać. Przedstawiony jest tu ogólny opis tego narzędzia wraz z uzasadnieniem i alternatywnymi rozwiązaniami na początku.

Tradycyjny, uniwersalny i ogólny sposób wywoływania klasy nadzędnej

W przykładach prezentowanych w tej książce preferowane jest wywoływanie metod klasy nadzędnej jawnie za pomocą ich nazw. Jest to tradycyjna technika w Pythonie, funkcjonująca tak samo w wersjach 2.x i 3.x, pozbawiona potencjalnych ograniczeń i złożoności. Jak widzieliśmy wcześniej, tradycyjny schemat wywoływania metody klasy nadzędnej wygląda następująco:

```
>>> class C:                                     # W Pythonie 2.x i 3.x
    def act(self):
        print('mielonka')

>>> class D(C):
    def act(self):
        C.act(self)          # Jawną nazwę klasy nadzędnej, argument self
        print('jajka')

>>> X = D()
>>> X.act()
mielonka
jajka
```

Powyższy sposób działa tak samo w wersjach 2.x i 3.x, zgodnie z modelem zwykłego mapowania wywołań metod dotyczy wszystkich form dziedziczenia i nie skutkuje nieprzewidzianym działaniem przeciążanych operatorów. Aby przekonać się, jakie to ma znaczenie, sprawdźmy, jak to się ma do funkcji `super`.

Podstawy i kompromisy użycia funkcji `super`

Ten podrozdział zarówno wprowadza do podstawowego trybu użycia funkcji `super` z jedną *instancją*, jak również opisuje jej wady w tej roli. Jak się przekonamy, w tym kontekście funkcja `super` działa zgodnie z oczekiwaniemi, ale nie różni się zbytnio od tradycyjnych wywołań, za to wykorzystuje nietypową semantykę i jest kłopotliwa w użyciu w wersji 2.x. Co gorsza, w miarę

powiększania się klas i wykorzystywania wielokrotnego dziedziczenia funkcja `super` może maskować problemy w kodzie i kierować wywołania w nieoczekiwany sposób.

Stara semantyka: magiczny obiekt pośredniczący w wersji 3.x

Wbudowana funkcja `super` pełni dwie role. Pierwsza i bardziej tajemnicza — kooperatywne protokoły kierowania wywołań w diamentowym drzewie wielokrotnego dziedziczenia (tak, dłuża nazwa!) — opiera się na algorytmie MRO z wersji 3.x. Została zapożyczona z języka Dylan i będzie omówiona w dalszej części tego rozdziału.

Rola, która nas teraz interesuje i jest częściej wykorzystywana i wymagana przez programistów Javy, to generyczne nadawanie nazw klasom nadrzędnym w drzewie dziedziczenia. Ma to na celu promowanie prostszej obsługi kodu i uniknięcie konieczności wpisywania długich ścieżek referencyjnych w wywołaniach klas nadrzędnych. Na pierwszy rzut okna wydaje się, że w Pythonie 3.x takie wywołanie pozwala osiągnąć zamierzony cel:

```
>>> class C:                                     # Tylko w Pythonie 3.x (patrz dalej forma z
   super w wersji 2.x)

       def act(self):
           print('mielonka')

>>> class D(C):
       def act(self):
           super().act(self)      # Generyczna referencja do klasy nadrzędnej
           z pominięciem self
           print('jajka')

>>> X = D()
>>> X.act()
mielonka
jajka
```

Powyższy kod działa poprawnie i minimalizuje konieczność wprowadzania zmian — jeżeli w przyszłości zmieni się klasa nadrzędna `D`, nie trzeba będzie modyfikować wywołań. Jednak jednym z największych mankamentów tego wywołania w wersji 3.x jest *wykorzystywana tu głęboka magia*. Jest ono podatne na zmiany, ale dzisiaj działa, sprawdzając stos wywołań, automatycznie lokalizuje argument `self` i klasę nadrzędną, które następnie łączy w parę w specjalnym *obiekcie pośredniczącym* kierującym późniejsze wywołania do odpowiedniej wersji metody w tej klasie. Ten opis brzmi skomplikowanie i dziwnie, bo faktycznie tak jest. W rzeczywistości wywołanie w takiej postaci nie działa w ogóle poza kontekstem metody klasy:

```
>>> super                         # „Magiczny” obiekt pośredniczący kierujący
   późniejszymi wywołaniami

<class 'super'>

>>> super()
SystemError: super(): no arguments

>>> class E(C):
       def method(self):          # Atrybut self jest stosowany niejawnie
           tylko w funkcji super

           proxy = super()         # Ta forma nie ma sensu poza metodą
```

```

        print(proxy)      # Pokazuje ukryty zazwyczaj obiekt
pośredniczący

proxy.act()          # Bez argumentów: niejawne wywołanie metody
klasy nadzędnej

>>> E().method()

<super: <class 'E'>, <E object>>

spam

```

Tak naprawdę semantyka tego wywołania nie jest podobna do niczego innego w Pythonie. Nie jest to ani metoda związana, ani niezwiązana, natomiast w jakiś sposób odnajduje klasę `self`, nawet jeżeli zostanie pominięta w wywołaniu. W drzewach jednej instancji klasa nadzędna jest dostępna z `self` poprzez ścieżkę `self.__class__.__bases__[0]`, jednak bardzo niejawna natura tego wywołania powoduje, że jest ono rzadko stosowane i stanowi całkowite zaprzeczenie stosowanej wszędzie zasady jawnego używania `self`. Oznacza to, że to wywołanie narusza podstawową regułę Pythona pojedynczego przypadku użycia i stosowaną od zawsze zasadę projektowania EIBTI (więcej informacji na temat tej reguły można uzyskać, wpisując polecenie `import this`).

Pułapka: beztroskie stosowanie wielokrotnego dziedziczenia

Rola funkcji `super`, nie licząc jej nietypowej semantyki, nawet w wersji 3.x dotyczy raczej drzewa pojedynczego dziedziczenia i zaczyna rodzić problemy, gdy tylko w tradycyjnie kodowanych klasach zaczyna być stosowane wielokrotne dziedziczenie. Wydaje się to istotnym ograniczeniem zakresu użycia funkcji. Wziawszy pod uwagę użyteczność mieszanych klas, wielokrotne dziedziczenie rozdzielnych i niezależnych klas nadzędnych jest w prawdziwym kodzie bardziej normą niż wyjątkiem. Wywołanie `super` wydaje się być receptą na zniszczenia wprowadzane w klasach beztrosko wykorzystujących podstawowy tryb bez dopuszczania o wiele bardziej subtelnych implikacji w drzewach wielokrotnego dziedziczenia.

Tę pułapkę ilustruje poniższy kod, który zaczyna swoje życie od szczęśliwego użycia funkcji `super` w trybie pojedynczego dziedziczenia w celu wywołania metody znajdującej się o jeden poziom wyżej od klasy C:

```

>>> class A:                                     # W Pythonie 3.x
    def act(self): print('A')

>>> class B:
    def act(self): print('B')

>>> class C(A):
    def act(self):
        super().act()                         # Funkcja super zastosowana w drzewie
                                                # pojedynczego dziedziczenia
>>> X = C()
>>> X.act()
A

```

Gdy jednak takie klasy zaczną rosnąć i wykorzystywać kilka klas nadzędnych, funkcja `super` stanie się podatna na błędy, a nawet bezużyteczna. Nie zgłosi wyjątku w drzewie wielokrotnego dziedziczenia, tylko w naturalny sposób wybierze *pierwszą z lewej* klasę nadzelną posiadającą metodę przeznaczoną do wywołania (z technicznego punktu widzenia pierwszą według algorytmu MRO), która może, ale nie musi być żądaną metodą:

```

>>> class C(A, B):           # Dodanie klasy B z taką samą metodą
    def act(self):
        super().act()  # Funkcja nie ulega awarii w wielokrotnym
dziedziczeniu, tylko po prostu wybiera metodę!
>>> X = C()
>>> X.act()
A
>>> class C(B, A):
    def act(self):
        super().act()  # Jeżeli klasa B jest wymieniona wcześniej,
metoda A.act() nie jest wywoływana!
>>> X = C()
>>> X.act()
B

```

Co gorsza, w ten sposób *po cichu maskowany* jest fakt, że w takim przypadku powinno się raczej *jawnie* wybierać klasę nadzczną w sposób opisany wcześniej w rozdziale tym i w poprzednich. Innymi słowy, funkcja `super` może ukrywać źródła często popełnianych błędów. Jest to tak często spotykany przypadek, że zostanie ponownie opisany w części poświęconej pułapkom. Skoro trzeba później użyć bezpośredniego wywołania, dlaczego nie można go również użyć wcześniej?

```

>>> class C(A, B):           # Tradycyjna forma
    def act(self):            # W tym miejscu prawdopodobnie trzeba być
bardziej jawnym
        A.act(self)          # Ta forma obsługuje zarówno pojedyncze,
jak i wielokrotne dziedziczenie
        B.act(self)          # Działa tak samo w wersjach Pythona 3.x i
2.x
>>> X = C()                 # Po co w ogóle stosować specjalny
przypadek funkcji super?
>>> X.act()
A
B

```

Jak za chwilę zobaczymy, tego rodzaju problem można rozwiązać, umieszczając wywołania `super` we wszystkich klasach w drzewie. Jednak jest to jeden z największych mankamentów funkcji: po co kodować ją w każdej klasie, skoro zazwyczaj nie jest potrzebna, a poprzednia, tradycyjna forma z jedną klasą zazwyczaj jest wystarczająca? Szczególnie w istniejącym kodzie (oraz nowym wykorzystującym istniejący) ten wymóg wydaje się wygórowany, a nawet nierealny.

Jak zobaczymy za chwilę, gdy zacznie się w ten sposób stosować wywołania w dziedziczeniu wielokrotnym, funkcja `super` może nie wywoływać klasy zgodnie z oczekiwaniami. Wywołania będą kierowane zgodnie z algorytmem MRO, który w zależności od miejsca użycia funkcji `super` może wywoływać metodę klasy, która w ogóle nie jest klasą nadzczną. Takie niejawne

wywołanie może bardzo urozmaicić sesję diagnostyczną! Bez głębokiej wiedzy, co oznacza użycie funkcji `super` po wprowadzeniu wielokrotnego dziedziczenia, lepiej jest nie stosować jej również w pojedynczym dziedziczeniu.

Opisana sytuacja w rzeczywistości nie jest tak abstrakcyjna, jak się może wydawać. Poniżej przedstawiony jest praktyczny przykład *PyMailGUI* wzięty z książki *Programming Python*. Przedstawione niżej bardzo typowe klasy wykorzystują wielokrotne dziedziczenie w celu zastosowania zarówno logiki aplikacji, jak i narzędzi wziętych z niezależnych, samodzielnych klas. Z tego powodu konstruktory klas nadzędnych muszą być jawnie wywoływanie bezpośrednio za pomocą nazw. W zakodowany tutaj sposób metoda `super().__init__()` uruchomi tylko jeden konstruktor. Dodanie wszędzie funkcji `super` w osobnych drzewach klas byłoby rozwiązaniem wymagającym większej pracy, nie byłoby prostsze i nie miało sensu w narzędziach przeznaczonych do swobodnego wdrażania u klientów, którzy mogą, ale nie muszą korzystać z funkcji `super`.

```
class PyMailServerWindow(PyMailServer, windows.MainWindow):
    "Tk z dodatkowym protokołem i wymieszany metodami"
    def __init__(self):
        windows.MainWindow.__init__(self, appname, srvrname)
        PyMailServer.__init__(self)

class PyMailFileWindow(PyMailFile, windows.PopupWindow):
    "Najwyższy poziom z dodatkowym protokołem i wymieszany metodami"
    def __init__(self, filename):
        windows.PopupWindow.__init__(self, appname, filename)
        PyMailFile.__init__(self, filename)
```

Kluczową kwestią tutaj jest to, że użycie funkcji `super` tylko w pojedynczym dziedziczeniu, w którym ma ona najbardziej uzasadnione zastosowanie, jest potencjalnym źródłem błędów i nieporozumień. Oznacza to bowiem, że programista musi znać dwa sposoby osiągnięcia tego samego celu, gdy tymczasem tylko jeden — jawnie i bezpośrednie wywołanie — wystarczy w każdej sytuacji.

Innymi słowy, jeżeli nie ma pewności, że w trakcie użytkowania programu nie zostanie do drzewa dziedziczenia dodana nowa klasa nadzędna, nie można używać funkcji `super` w trybie pojedynczego dziedziczenia bez uprzedniego zrozumienia i uwzględnienia jego o wiele bardziej wyrafinowanej roli w wielokrotnym dziedziczeniu. Ten temat będzie opisany później, ale nie można go pominąć, jeżeli stosuje się funkcję `super`.

Z bardziej praktycznego punktu widzenia też nie jest oczywiste, że minimalizacja nakładu pracy związanego z utrzymaniem kodu, co jest jednym z celów funkcji `super`, w pełni uzasadnia jej obecność. W praktyce nazwy klas nadzędnych w nagłówkach są rzadko zmieniane, a jeżeli już, zazwyczaj dotyczy to bardzo małej liczby wywołań klas nadzędnych wewnątrz danej klasy. Należy rozważyć następującą kwestię: jeżeli w przyszłości zostanie dodana klasa nadzędna, która nie będzie wykorzystywała funkcji `super` (jak w poprzednim przykładzie), należy ją albo opakować w obiekt pośredniczący, albo zmienić wszystkie wywołania `super` w klasie tak, aby wykorzystywały tradycyjny schemat jawnego wywoływania za pomocą nazw. Jest to zadanie utrzymaniowe, które wydaje się równie prawdopodobne, ale bardziej podane na błędy, jeżeli zacznie się polegać na magii funkcji `super`.

Ograniczenie: przeciążanie operatorów

Jak krótko wspomniano w podręczniku do Pythona, funkcja `super` nie działa w pełni poprawnie, jeżeli stosowane są metody `__X__` przeciążające operatory. Po przeanalizowaniu poniższego

kodu można zauważyc, że bezpośrednie nazwane wywołania w klasie nadzędnej metod przeciążających operatory działają poprawnie, natomiast użycie wyniku funkcji `super` w wyrażeniu uniemożliwia właściwe skierowanie metody przeciążającej:

```
>>> class C:                                     # W Pythonie 3.x
        def __getitem__(self, ix):                 # Metoda przeciążająca indeks
            print('C index')

>>> class D(C):
    rozszerzenia
        def __getitem__(self, ix):                 # Ponowna definicja w celu
            print('D index')
            C.__getitem__(self, ix)                # Tradycyjne wywołanie działa
            super().__getitem__(ix)                # Bezpośrednie wywołanie po
                                              # nazwie też działa
            super().__getitem__                   # Jednak operatory nie
                                              # działają! (__getattribute__)

>>> X = C()
>>> X[99]
C index

>>> X = D()
>>> X[99]
D index
C index
C index

Traceback (most recent call last):
  File "", line 1, in
    File "", line 6, in __getitem__
      TypeError: 'super' object is not subscriptable
```

Takie działanie jest skutkiem tej samej zmiany klas w nowym stylu (i w wersji 3.x), która została opisana wcześniej w tym rozdziale (patrz podrozdział „Pomijanie instancji we wbudowanych operacjach przy pobieraniu atrybutów”). Ponieważ obiekt pośredniczący zwracany przez funkcję `super` wykorzystuje metodę `__getattribute__` do przechwytywania i kierowania późniejszych wywołań metod, nie może przechwytywać metod `X` wywoływanych automatycznie przez wbudowane operacje, m.in. wyrażenia, ponieważ rozpoczynają one swoje wyszukiwania w klasie, a nie w instancji. Może się to wydawać mniej dotkliwym ograniczeniem niż w przypadku wielokrotnego dziedziczenia, ale operatory powinny działać tak samo jak odpowiadające im wywołania metod, szczególnie wbudowanych, jak w tym przypadku. Brak takiej obsługi wprowadza kolejny wyjątek, który użytkownicy funkcji `super` muszą rozważyć, a potem o nim pamiętać.

W Pythonie atrybut `self` jest jawnym mieszaniną klas i przeciążaniem operatorów powszechnie stosowane, a nazwy klas nadzędnych zmieniane są rzadko. Ponieważ funkcja `super` wprowadza do języka przypadek specjalny — o dziwnej semantyce, ograniczonym zakresie, surowych wymaganiach i kwestionowanych korzyściach — dla większości programistów lepszym

rozwiązaniem będzie pozostanie przy tradycyjnym, szerzej stosowanym schemacie wywołań. Funkcja `super` ma wprawdzie kilka zaawansowanych zastosowań, które za chwilę poznamy, jednak są one zbyt tajemnicze, aby stały się obowiązkowym repertuarzem każdego programisty.

Różnice w użyciu w wersji 2.x: rozbudowane wywołania

Jeżeli używamy wersji 2.x, czytając tę książkę poświęconą obu wersjom, pamiętajmy, że techniki wykorzystujące funkcję `super` nie są uniwersalne. Ich formy są różne w wersjach 2.x i 3.x i dotyczy to nie tylko klas zwykłych i w nowym stylu. W wersji 2.x jest to zupełnie inne narzędzie, którego nie można używać w wersji 3.x w prostszej formie.

Aby to wywołanie działało w Pythonie 2.x, trzeba przede wszystkim stosować *klasy w nowym stylu*. Jednak nawet wtedy trzeba jawnie przekazywać funkcji `super` nazwę pośredniej klasy i atrybut `self`, przez co wywołanie staje się tak skomplikowane i rozbudowane, że w większości przypadków prawdopodobnie lepiej jest całkowicie z niego zrezygnować i po prostu jawnie podawać nazwę klasy nadchodzącej zgodnie z tradycyjnym schematem wywołań (dla zwięzości opisu pozostawię czytelnikowi zastanowienie się, co oznacza zmiana nazwy klasy pod kątem utrzymania kodu wykorzystującego funkcję `super` w wersji 2.x!).

```
>>> class C(object):                                # W Pythonie 2.x: tylko w klasach w
   nowym stylu

   def act(self):
       print('mielonka')

>>> class D(C):
   def act(self):
       super(D, self).act()                      # Wersja 2.x: inny, zbyt
   skomplikowany format wywołania
       print('jajka')                          # Nazwa „D”, podobnie jak „C”, może
   się zmienić!

>>> X = D()
>>> X.act()
mielonka
jajka
```

Choć w wersji 3.x można w celu uzyskania kompatybilności używać wywołania w formie właściwej dla wersji 2.x, jest to zbyt kłopotliwe, natomiast rozsądna forma w wersji 3.x nie ma zastosowania w wersji 2.x:

```
>>> class D(C):
   def act(self):
       super().act()                         # Prostszy format wywołania w wersji
   3.x nie działa w wersji 2.x
       print('eggs')

>>> X = D()
>>> X.act()
TypeError: super() takes at least 1 argument (0 given)
```

Z drugiej strony tradycyjna forma wywołania z jawną nazwą klasy działa w wersji 2.x zarówno w przypadku zwykłych klas, jak i klas w nowym stylu, identycznie jak w wersji 3.x:

```

>>> class D(C):
        def act(self):
            C.act(self)                      # Jednak tradycyjny sposób jest
uniwersalny,
            print('jajka')                  # a w wersji 2.x często prostszy

>>> X = D()
>>> X.act()
mielonka
jajka

```

Po co więc stosować technikę, która działa tylko w wybranych kontekstach, zamiast innej, o znacznie szerszym zastosowaniu? W kolejnych podrozdziałach zawarte są argumenty przemawiające za stosowaniem funkcji `super`.

Zalety funkcji super: zmiany drzewa i kierowania metod

Opisałem mankamenty funkcji `super`, jednak muszę się przyznać, że miałem pokusę, aby używać tego wywołania w kodzie, który działałby tylko w wersji 3.x, i w którym stosowana byłaby dłuża ścieżka referencyjna prowadząca przez pakiet do klasy nadzędnej (co miałoby negatywny wpływ na zwięzłość kodu). Trzeba jednak przyznać, że funkcja `super` jest przydatna w kilku przypadkach, z których najważniejsze warto w tym miejscu przedstawić:

- *Modyfikacja drzewa klas w trakcie działania programu.* Jeżeli w trakcie działania programu trzeba zmienić klasę nadzędną, nie można tego zrobić, wpisując na stałe jej nazwę w wyrażeniu wywołania. Można za to zmienić wywołanie za pomocą funkcji `super`.
- Ten przypadek jest jednak wyjątkowo rzadki, a poza tym w tym kontekście zazwyczaj można zastosować inną technikę.
- *Kooperatywne kierowanie metod w drzewie wielokrotnego dziedziczenia.* Jeżeli drzewo wielokrotnego dziedziczenia muszą kierować metody o takich samych nazwach, ale należące do różnych klas, wtedy funkcja `super` oferuje protokół do uporządkowanego kierowania wywołań.

Z drugiej strony drzewo musi opierać się na klasach uporządkowanych za pomocą algorytmu MRO. Jest to narzędzie skomplikowane samo w sobie, a jego użycie do rozwiązania głównego problemu jest sztuczne. Aby rozwiązanie było skuteczne, drzewo musi być zakodowane i rozszerzone tak, aby funkcja `super` była wykorzystywana w każdej metodzie. Tego rodzaju kierowanie można zazwyczaj zaimplementować w inny sposób (np. wykorzystując stan instancji).

Jak wspomniałem wcześniej, funkcji `super` można również używać do generycznego wybierania klasy nadzędnej, o ile domyślne działanie algorytmu MRO ma sens. Jednak w tradycyjnym kodzie jest preferowane, a nawet wymagane, jawnie odwoływanie się do klasy nadzędnej za pomocą nazwy. Co więcej, nawet w uzasadnionych przypadkach rzadko stosuje się funkcję `super`; na tyle rzadko, że niektórzy programiści taktują je jako akademickie ciekawostki. Wymienione wyżej dwa przypadki są jednak najczęściej podawanymi uzasadnieniami, dlatego przyjrzyjmy się teraz każdemu z nich.

Zmiana klasy w trakcie działania programu a funkcja super

Jeżeli w trakcie działania programu trzeba zmieniać klasy nadrzędne, nie można wpisywać na stałe ich nazw w metodach klas podrzędnych. Na szczęście funkcja `super` może dynamicznie wyszukiwać bieżącą klasę nadrzędną. Niemniej jednak ten przypadek jest w praktyce spotykany zbyt rzadko, aby uzasadniał użycie funkcji `super`, a ponadto w sytuacjach wyjątkowych w razie potrzeby można go zaimplementować w inny sposób. Poniższy kod ilustruje dynamiczną zmianę klasy nadrzędnej C poprzez modyfikację krotki `__bases__` w klasie podrzędnej w wersji 3.x:

```
>>> class X:
        def m(self): print('X.m')

>>> class Y:
        def m(self): print('Y.m')

>>> class C(X):                               # Zaczynamy od dziedziczenia klasy
X
        def m(self): super().m()                 # Tutaj nie można wpisać na stałe
nazwy klasy
>>> i = C()
>>> i.m()
X.m
>>> C.__bases__ = (Y,)                      # Zmiana klasy nadrzędnej w
trakcie działania kodu!
>>> i.m()
Y.m
```

Ten sposób działa (i też wykorzystuje tajemną magię, jaką jest zmiana atrybutu `__class__` instancji), ale jest stosowany wyjątkowo rzadko. Co więcej, istnieją inne sposoby osiągnięcia tego celu. Prawdopodobnie najprostszym z nich jest wywołanie pośrednie wykorzystujące krotkę klasy nadrzędnej. Jest to szczególny kod, ale tylko w tym szczególnym przypadku (choć nie jest bardziej szczególny od niejawnego kierowania wywołań za pomocą algorytmu MRO):

```
>>> class C(X):
        def m(self): C.__bases__[0].m(self)    # Specjalny kod w
szczególnym przypadku
>>> i = C()
>>> i.m()
X.m
>>> C.__bases__ = (Y,)                      # Ten sam efekt bez użycia
funkcji super
>>> i.m()
Y.m
```

Zważywszy, że istnieją alternatywne rozwiązania, taki pojedynczy przypadek nie uzasadnia użycia funkcji `super`, choć w bardziej złożonych drzewach może pojawić się inne uzasadnienie

— oparte na kolejności przeszukiwania drzewa określonej przez MRO, a nie fizycznych łączach klasy nadzędnej.

Kooperatywne kierowanie metod w drzewie wielokrotnego dziedziczenia

Drugi z wymienionych wcześniej przypadków szczególnych jest głównym powodem uzasadniającym stosowanie funkcji `super`. Został zapożyczony z innych języków (głównie z Dylana), w których występuje on częściej niż w typowym kodzie w Pythonie. Zazwyczaj dotyczy diamentowego wielokrotnego dziedziczenia opisanego wcześniej w tym rozdziale i pozwala kooperatywnym i zgodnym klasom na kierowanie wywołań do *metod o takich samych nazwach* w jednolity sposób między implementacjami różnych klas. Takie rozwiązanie, stosowane w spójny sposób, przede wszystkim upraszcza protokół kierowania wywołań konstruktorów, które zazwyczaj mają wiele implementacji.

W tym trybie każde wywołanie `super` wybiera metodę z następnej klasy w kolejności MRO. Ten proces wybiera pierwszą klasę następującą po klasie wywołującej, posiadającą żądany atrybut. Opisany wcześniej algorytm MRO określa ścieżkę, którą Python podąża podczas dziedziczenia klas w nowym stylu. Ponieważ liniowa kolejność MRO zależy od tego, której klasy dotyczy atrybut `self`, kolejność wywoływanych metod określona przez funkcję `super` może być różna w różnych drzewach, a każda klasa jest odwiedzana tylko raz, o ile wszystkie wykorzystują do wywoływania funkcję `super`.

Ponieważ każda klasa w wersji 3.x (i klasa w nowym stylu w wersji 2.x) tworzy diamentową strukturę z klasą `object`, zastosowanie tego rozwiązania jest szersze, niż można się było spodziewać. W rzeczywistości w niektórych przykładach demonstrujących mankamenty funkcji `super` w drzewach wielokrotnego dziedziczenia można stosować to wywołanie, aby osiągnąć zamierzony cel. Wtedy jednak funkcja ta musiałaby być stosowana *uniwersalnie* w drzewie klas, aby można było przekazywać łańcuch wywołań metod. Jest to ważny wymóg, który trudno jest spełnić w większości istniejących i nowych kodów.

Podstawy: kooperatywne wywołanie funkcji `super` w akcji

Sprawdźmy, co ta rola oznacza w kodzie. W podrozdziale tym i w następnych dowiemy się, jak działa funkcja `super`, oraz poznamy narzucone przez nią kompromisy. Przeanalizujmy najpierw poniższe, zakodowane w tradycyjny sposób klasę (jak zwykle nieco skondensowane, aby zaoszczędzić miejsca):

```
>>> class B:
    def __init__(self): print('B.__init__')    # Oddzielne gałęzie
drzewa

>>> class C:
    def __init__(self): print('C.__init__')

>>> class D(B, C): pass
# Domyślne uruchomienie

>>> x = D()
metody pierwszej z lewej

B.__init__
```

W tym przypadku gałęzie drzewa klas nadzędnych są *rozdzielone* (nie mają jawnego wspólnego przodka), więc łączące je klasy podrzędne muszą odwoływać się do nich za pomocą nazw. Jest to często spotykana sytuacja, w której nie można bezpośrednio użyć funkcji `super` bez zmieniania kodu:

```
>>> class D(B, C):
        def __init__(self):          # Tradycyjna forma
            B.__init__(self)         # Wywołanie klas nadzędnych za
pomocą nazw
            C.__init__(self)

>>> x = D()
B.__init__
C.__init__
```

Jednak w drzewie *diametrowym* wywołania wykorzystujące *jawne nazwy* mogą domyślnie uruchamiać więcej niż raz metody w klasie znajdującej się na najwyższym poziomie. Można temu zapobiec, stosując dodatkowe protokoły (np. znaczniki stanu w instancji):

```
>>> class A:
        def __init__(self): print('A.__init__')

>>> class B(A):
        def __init__(self): print('B.__init__'); A.__init__(self)

>>> class C(A):
        def __init__(self): print('C.__init__'); A.__init__(self)

>>> x = B()
B.__init__
A.__init__

>>> x = C()                      # Każda funkcja super działa niezależnie
C.__init__
A.__init__

>>> class D(B, C): pass          # Wciąż wywoływana jest metoda pierwsza z
lewej

>>> x = D()
B.__init__
A.__init__

>>> class D(B, C):
        def __init__(self):      # Tradycyjna forma
            B.__init__(self)    # Wywołanie obu klas nadzędnych za
pomocą nazw
            C.__init__(self)

>>> x = D()                      # Jednak wywołuje to klasę A dwukrotnie!
B.__init__
A.__init__
```

```
C.__init__
```

```
A.__init__
```

Jeżeli natomiast wszystkie klasy używają funkcji `super` lub są odpowiednio zmuszone przez obiekt pośredniczący, aby zachowywały się tak, jakby tej funkcji używały, wtedy wywołania metod są kierowane do klasy zgodnie z algorytmem MRO, dzięki czemu metody klasy na najwyższym poziomie są wywoływane tylko raz:

```
>>> class A:
    def __init__(self): print('A.__init__')

>>> class B(A):
    def __init__(self): print('B.__init__'); super().__init__()

>>> class C(A):
    def __init__(self): print('C.__init__'); super().__init__()

>>> x = B() # Uruchamia B.__init__, klasa A jest zgodnie z MRO następną
          klasą nadzrędną w atrybucie self klasy B

B.__init__
A.__init__

>>> x = C()

C.__init__
A.__init__

>>> class D(B, C): pass

>>> x = D() # Uruchamia B.__init__, klasa C jest zgodnie z MRO następną
          klasą nadzrędną w atrybucie self klasy D!

B.__init__
C.__init__
A.__init__
```

Prawdziwą magią, która się tutaj kryje, jest liniowa lista MRO utworzona dla klasy `self`. Ponieważ każda klasa pojawia się na liście tylko raz, a funkcja `super` wywołuje następną klasę z listy, zapewniony jest uporządkowany łańcuch wywołań, w którym każda klasa jest odwiedzana tylko raz. Ważne jest to, że *następna* klasa po B zgodnie z MRO jest inna niż `self`. Dla instancji B jest to klasa A, a dla instancji D klasa C, zgodnie z kolejnością uruchamiania konstruktorów:

```
>>> B.__mro__
(<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)

>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>)
```

Algorytm MRO został zaprezentowany wcześniej w tym rozdziale. Wybierając następną klasę w sekwencji MRO, funkcja `super` w metodzie klasy *eskaliuje* wywołanie w drzewie, o ile wszystkie klasy robią to samo. W tym trybie funkcja `super` nie musi wybierać klasy nadzędnej w ogóle. Wybiera następną w liniowej kolejności MRO, która może być klasą *bliźniaczą*, a nawet *niższą* w drzewie klas krewną dla danej instancji. Więcej przykładów ścieżki, którą podąża wywołanie

`super`, szczególnie w drzewach innych niż diamentowe, znajduje się w podrozdziale „Śledzenie algorytmu MRO”.

Poprzednie rozwiążanie działa i nawet na pierwszy rzut oka wygląda pomysłowo. Jednak możliwości jego zastosowania części programistów mogą wydawać się ograniczone. W większości programów w Pythonie nie są wykorzystywane niuanse diamentowych drzew wielokrotnego dziedziczenia (w rzeczywistości wielu programistów, których spotkałem, nie wie nawet, co ten termin oznacza!). Co więcej, funkcja `super` najbardziej dotyczy przypadków pojedynczego dziedziczenia i kooperatywnej struktury diamentowej. Zatem w przypadkach rozdzielnych struktur, innych niż diamentowe, funkcja ta wydaje się być niepotrzebna, ponieważ można wybiórczo lub niezależnie wywoływać metody klas nadzędnych. Nawet kooperatywnymi, niedomenowymi drzewami można zarządzać, wykorzystując inne sposoby dające programistom większą kontrolę niż w przypadku automatycznego algorytmu MRO. Jednak aby obiektywnie ocenić to narzędzie, musimy się mu bliżej przyjrzeć.

Ograniczenie: wymóg zakotwiczenia łańcucha wywołań

Wywołanie `super` cechuje się złożonością, która może nie być widoczna na pierwszy rzut oka, a nawet może być uznana za pewną funkcjonalność. Na przykład kolejność MRO może być wykorzystywana nawet w przypadkach, gdy struktura diamentowa jest niejawną, ponieważ w wersji 3.x *wszystkie* klasy (a w wersji 2.x jawnie zadeklarowane klasy w nowym stylu) automatycznie pochodzą od klasy `object`. W poniższym kodzie konstruktory w niezależnych klasach są wywoływane automatycznie.

```
>>> class B:
        def __init__(self): print('B.__init__'); super().__init__()

>>> class C:
        def __init__(self): print('C.__init__'); super().__init__()

>>> x = B()                      # Klasa object jest niejawną klasą na końcu MRO
        B.__init__

>>> x = C()
        C.__init__

>>> class D(B, C): pass          # Dziedziczy B.__init__, jednak MRO klasy B jest
                                # inny niż D
        # Wywołuje B.__init__, klasa C jest zgodnie z
        # MRO następną klasą nadzędną w atrybucie self klasy D
        B.__init__
        C.__init__
```

Z technicznego punktu widzenia ten model kierowania wywołań wymaga, aby metoda wywoływana przez funkcję `super` istniała i miała tę samą sygnaturę argumentów w całym drzewie klas, a ponadto w każdym wystąpieniu oprócz ostatniego sama korzystała z funkcji `super`. Powyższy przykład działa tylko dlatego, że niejawną klasą nadzędną `object` na końcu MRO wszystkich trzech klas posiada kompatybilną metodę `__init__` spełniającą powyższe warunki:

```
>>> B.__mro__
(<class '__main__.B'>, <class 'object'>)
>>> D.__mro__
```

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>)
```

Tutaj dla instancji D następną klasą w kolejności MRO po klasie B jest klasa C, po której następuje obiekt, którego metoda `__init__` niejawnie przyjmuje wywołanie z klasy C i kończy łańcuch. Dzięki temu metoda klasy B wywołuje klasę C, która kończy łańcuch w klasie object, mimo że nie jest to klasa nadzędna dla klasy B.

W rzeczywistości jest to nietypowy przypadek, być może nawet *szczęśliwy*. W większości sytuacji w klasie object nie istnieją żadne odpowiednie ustawienia i spełnienie oczekiwania tego modelu może nie być takim trywialnym zadaniem. Większość drzew wymaga jawnej, a *nawet dodatkowej* klasy nadzędnej, która mogłaby pełnić rolę zakotwiczenia, jak tutaj robi to klasa object, aby akceptować wywołania, ale ich dalej nie przesyłać. Inne drzewa mogą wymagać starannego projektowania, aby spełnić ten wymóg. Co więcej, jeżeli Python tego nie zoptymalizuje, wywołanie klasy object (lub innego zakotwiczenia) domyślnie odbywa się na końcu łańcucha i może dodatkowo pogorszyć *wydajność kodu*.

Natomiast bezpośrednie wywołania w takich przypadkach nie wymagają specjalnego kodowania ani nie pogarszają wydajności. Za to kierowanie jest bardziej jawne i bezpośrednie:

```
>>> class B:  
    def __init__(self): print('B.__init__')  
  
>>> class C:  
    def __init__(self): print('C.__init__')  
  
>>> class D(B, C):  
    def __init__(self): B.__init__(self); C.__init__(self)  
  
>>> x = D()  
B.__init__  
C.__init__
```

Zakresy: model „wszystko lub nic”

Należy pamiętać, że tradycyjne klasy, które nie wykorzystują funkcji `super` w opisanej roli, nie mogą być bezpośrednio stosowane w kooperatywnych drzewach, ponieważ nie przekazują wywołań zgodnie z łańcuchem MRO. Możliwe jest włączenie takich klas do obiektów pośredniczących, które opakowują oryginalny obiekt i dodają wymagane wywołanie `super`, ale oznacza to wymóg dodatkowego kodowania i pogorszenie wydajności modelu. Jest to poważna przeszkoda, jeżeli istniejący kod składa się z milionów wierszy i *nie wykorzystuje* funkcji `super`.

Sprawdźmy na przykład, co się stanie, jeżeli któraś z klas pominie funkcję `super` i nie przekaże wywołania wzduż łańcucha, przez co zakończy go przedwcześnie. Funkcja `super`, podobnie jak atrybut `__slots__`, jest funkcjonalnością typu „wszystko lub nic”.

```
>>> class B:  
    def __init__(self): print('B.__init__'); super().__init__()  
  
>>> class C:  
    def __init__(self): print('C.__init__'); super().__init__()  
  
>>> class D(B, C):  
    def __init__(self): print('D.__init__'); super().__init__()  
  
>>> X = D()
```

```

D.__init__
B.__init__
C.__init__
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'object'>)
# Co się stanie, gdy używa się klasy, która nie wywołuje funkcji super?
>>> class B:
        def __init__(self): print('B.__init__')
>>> class D(B, C):
        def __init__(self): print('D.__init__'); super().__init__()
>>> X = D()
D.__init__
B.__init__      # Jest to narzędzie typu „wszystko lub nic”

```

Spełnienie wymogu eskalowania wywołań może nie być tak proste jak bezpośrednie wywołanie za pomocą nazwy, o czym można zapomnieć, ale co wymaga wpisywania całego kodu wykorzystywanego przez klasy. Jak wspomniałem, można dostosować klasę taką jak B, dziedzicząc ją po klasie *pośredniczącej*, która zawiera instancje klasy B. Jednak takie rozwiązanie wydaje się sztuczne w odniesieniu do celów programu, wprowadza dodatkowe wywołanie do każdej opakowanej metody i może powodować opisane wcześniej problemy typowe dla klas w nowym stylu związane z interfejsami i wbudowanymi operacjami. Z tego powodu jawi się jako niezwykły, a nawet oszałamiający, *dodatkowy wymóg kodowania* w modelu, który ma upraszczać kod.

Elastyczność założenia dotyczącego kolejności wywołań

Kierowanie wywołań zakłada również, że wszystkie wywołania metod będą naprawdę przechodzić przez wszystkie klasy zgodnie z MRO, co może, ale nie musi być zgodne z wymaganiami. Wyobraźmy sobie na przykład, że bez względu na inne potrzeby porządkowania dziedziczenia w poniższym kodzie wersja danej metody w klasie C musi w pewnych kontekstach być wywoływana przed klasą B. Jeżeli MRO mówi inaczej, wracamy do tradycyjnych wywołań, co może kolidować z użyciem funkcji super, a w poniższym kodzie powodować dwukrotne wywołanie metody klasy C.

```

# Co się stanie, jeżeli wymagana kolejność wywołań będzie inna niż MRO?
>>> class B:
        def __init__(self): print('B.__init__'); super().__init__()
>>> class C:
        def __init__(self): print('C.__init__'); super().__init__()
>>> class D(B, C):
        def __init__(self): print('D.__init__'); C.__init__(self);
B.__init__(self)
>>> X = D()
D.__init__

```

```
C.__init__  
B.__init__  
C.__init__
```

Analogicznie, jeżeli niektóre metody mają w ogóle nie być wywoływane, wtedy automatyczna ścieżka funkcji `super` nie jest stosowana tak bezpośrednio jak jawne wywołania, przez co trudno jest uzyskać ściślejszą kontrolę nad procesem kierowania wywołań. W rzeczywistych programach zawierających wiele metod, zasobów i zmiennych stanu to całkiem prawdopodobny scenariusz. Można wprawdzie zmienić kolejność klas nadzorowanych w klasie D dla tej metody, ale nie spełni się wtedy innych wymagań.

Dostosowywanie: zastąpienie metody

Wymóg uniwersalnego wdrożenia funkcji `super` może skutecznie utrudniać w pojedynczej klasie zamianę (przesłonięcie) odziedziczonej metody. Nieprzekazanie wywołania wyżej za pomocą funkcji `super` — celowe w tym przypadku — sprawdza się w samej klasie, ale może przerwać łańcuch wywołań w drzewie i uniemożliwić uruchamianie innych metod. Przeanalizujmy poniższy kod:

```
>>> class A:  
        def method(self): print('A.method'); super().method()  
  
>>> class B(A):  
        def method(self): print('B.method'); super().method()  
  
>>> class C:  
        def method(self): print('C.method')      # Brak funkcji super:  
musi kotwiczyć łańcuch!  
  
>>> class D(B, C):  
        def method(self): print('D.method'); super().method()  
  
>>> X = D()  
  
>>> X.method()  
D.method  
B.method  
A.method      # Automatyczne wywołanie wszystkich metod zgodnie z MRO  
C.method
```

Zastąpienie metody zakłóca tutaj model funkcji `super` i prowadzi nas do tradycyjnej formy:

```
# Co, jeśli klasa musi całkowicie zastąpić domyślną funkcję super?  
  
>>> class B(A):  
        def method(self): print('B.method')  # Usunięcie super w celu  
zastąpienia metody klasy A  
  
>>> class D(B, C):  
        def method(self): print('D.method'); super().method()  
  
>>> X = D()  
  
>>> X.method()
```

```

D.method
B.method          # Zastąpienie przerywa jednak łańcuch wywołań...
>>> class D(B, C):
    def method(self): print('D.method'); B.method(self);
C.method(self)
>>> D().method()
D.method
B.method
C.method          # Wracamy do jawnych wywołań

```

Jak poprzednio problem z założeniami polega na tym, że zakłada się różne rzeczy! Choć założenie o uniwersalnym kierowaniu może być uzasadnione w przypadku konstruktorów, koliduje jednak z podstawowymi regułami programowania obiektowego, tj. nieograniczonym dostosowywaniem klas podrzędnych. Może to oznaczać ograniczenie użycia funkcji super do konstruktorów, ale nawet one mogą czasami wymagać zastąpienia, a to wprowadza dziwny wymóg szczególnego przypadku dla jednego konkretnego kontekstu. Narzędzie, które można stosować tylko w odniesieniu do pewnej kategorii metod, zważywszy na komplikacje, które wprowadza, może się wydawać niepotrzebne, a nawet szkodliwe.

Sprzęganie: zastosowanie w mieszaniu klas

Mówiąc, że funkcja super wybiera *następną klasę* w kolejności MRO, w rzeczywistości mamy na myśli klasę, która *implementuje wymaganą metodę*. Funkcja pomija kolejne klasy, aż znajdzie tę o wymaganej nazwie. Ma to znaczenie w przypadku niezależnych, mieszanych klas, które mogą być dowolnie dodawane do drzew klienckich. Bez pomijania mieszane klasy nie działałyby wcale — porzuciłyby łańcuch wywołań dowolnych metod klienckich i nie mogłyby wykorzystywać swoich własnych funkcji super i działać zgodnie z oczekiwaniemi.

Na przykład w poniższych niezależnych odgałęzieniach przekazywane jest wywołanie metody klasy C, mimo że klasa Mixin, następna w kolejności MRO w instancji klasy C, nie definiuje metody o zadanej nazwie. Dopóki zbiory nazw metod są rozłączne, ten sposób po prostu działa — łańcuchy wywołań każdej gałęzi mogą istnieć niezależnie od siebie:

```

# Klasy mieszane działają tylko w przypadku rozłącznych zbiorów metod
>>> class A:
    def other(self): print('A.other')
>>> class Mixin(A):
    def other(self): print('Mixin.other'); super().other()
>>> class B:
    def method(self): print('B.method')
>>> class C(Mixin, B):
    def method(self): print('C.method'); super().other();
super().method()
>>> C().method()
C.method
Mixin.other

```

```
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>)
```

I analogicznie: mieszanie w inny sposób nie przerywa łańcucha wywołań. Na przykład w poniższym kodzie, mimo że klasa B nie definiuje metody `other`, gdy jest wywoływana w klasie C, klasy robią to później w kolejności MRO. W rzeczywistości łańcuch wywołań działa nawet wtedy, gdy jedno z odgałęzień w ogóle nie wykorzystuje funkcji `super`. Jeżeli tylko metoda jest zdefiniowana gdzieś dalej w kolejności MRO, jej wywołanie działa:

```
>>> class C(B, Mixin):
        def method(self): print('C.method'); super().other();
    super().method()
>>> C().method()
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.Mixin'>,
<class '__main__.A'>, <class 'object'>)
```

Jest tak również w drzewie *diamentowym* — rozdzielne zbiory metod są kierowane zgodnie z oczekiwaniemi, nawet jeżeli nie są zaimplementowane w każdym z oddzielnych odgałęzień, ponieważ wybierana jest następna klasa w kolejności MRO posiadająca daną metodę. W rzeczywistości stanowią one równoważne konteksty, ponieważ MRO zawiera te same klasy, a klasy podzielne zawsze pojawiają się w MRO przed swoimi klasami nadzorowanymi. Na przykład w poniższym kodzie wywołana metoda `other` w klasie `Mixin` jest znajdowana w klasie A, mimo że kolejną klasą po `Mixin` jest w MRO klasa B (wywołanie metody `method` w klasie C też działa z tych samych powodów):

```
# Jawną strukturę diamentową też działa
>>> class A:
        def other(self): print('A.other')
>>> class Mixin(A):
        def other(self): print('Mixin.other'); super().other()
>>> class B(A):
        def method(self): print('B.method')
>>> class C(Mixin, B):
        def method(self): print('C.method'); super().other();
    super().method()
>>> C().method()
```

```

C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.B'>,
<class '__main__.A'>, <class 'object'>)
# Inna kolejność klas mieszanych też działa
>>> class C(B, Mixin):
        def method(self): print('C.method'); super().other();
    super().method()
>>> C().method()
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.Mixin'>,
<class '__main__.A'>, <class 'object'>)

```

Wciąż jednak uzyskany efekt niczym się nie różni od bezpośrednich wywołań za pomocą nazw, które również działają w tym przypadku niezależnie od kolejności klas nadzędnych i od tego, czy drzewo jest diamentowe, czy nie. W takim przypadku kolejność MRO wydaje się niepewna, skoro tradycyjna forma jest prostsza, bardziej jawną, oferującą większą kontrolę i elastyczność:

```

# Wywołania też tutaj działają: jawnie są lepsze niż niejawne
>>> class C(Mixin, B):
        def method(self): print('C.method'); Mixin.other(self);
    B.method(self)
>>> X = C()
>>> X.method()
C.method
Mixin.other
A.other
B.method

```

Co ważne, do tej pory w tym przykładzie przyjęte było założenie, że nazwy metod są rozłączne w swoich gałęziach. Kolejność wysyłania metod o takich samych nazwach w diamentowym drzewie takim jak tutaj może być znacznie mniej przypadkowa, ponieważ nie ma możliwości, aby klasa kliencka unieważniła zamiar wywołania funkcji super. Wywołanie metody method w

klasie Mixin uruchamia wersję metody w klasie A zgodnie z oczekiwaniami, chyba że będzie zmieszana w drzewie, w którym porzucany jest łańcuch wywołań:

```
# W przypadku nierozdzielnych metod: funkcja super tworzy nadmierne
# sprzeżenie

>>> class A:
    def method(self): print('A.method')

>>> class Mixin(A):
    def method(self): print('Mixin.method'); super().method()

>>> Mixin().method()
Mixin.method
A.method

>>> class B(A):
    def method(self): print('B.method')      # Funkcja super wywoła
                                               klasę A po B

>>> class C(Mixin, B):
    def method(self): print('C.method'); super().method()

>>> C().method()
C.method
Mixin.method
B.method      # Klasę A pomijamy tylko w tym kontekście!
```

Może się zdarzyć, że klasa B nie będzie redefiniować metody (narażamy się wtedy na ogólne problemy związane z wielokrotnym dziedziczeniem), jednak nie musi to zakłócać mieszanego. W takich wypadkach *bezpośrednie wywołania* dają większą kontrolę i pozwalają mieszanym klasom na znacznie większą niezależność w kontekście użycia:

```
# Bezpośrednie wywołania są odporne na kontekst użycia

>>> class A:
    def method(self): print('A.method')

>>> class Mixin(A):
    def method(self): print('Mixin.method'); A.method(self)      #
                                               Klasa C nie ma znaczenia

>>> class C(Mixin, B):
    def method(self): print('C.method'); Mixin.method(self)

>>> C().method()
C.method
Mixin.method
A.method
```

Co więcej, jeżeli klasy mieszane będą bardziej *samodzielne*, bezpośrednie wywołania będą minimalizować *sprząganie komponentów*, które zwiększa złożoność programu. Jest to

podstawowa zasada programowania, najwyraźniej negowana przez funkcję `super` i kontekstowy model kierowania metod.

Dostosowywanie: wymóg takich samych argumentów

Na koniec należy również rozważyć konsekwencje użycia funkcji `super`, gdy argumenty metody różnią się w zależności od klasy. Ponieważ twórca klasy nie wie, którą wersję metody będzie wywoływała funkcja `super` (w rzeczywistości może to zależeć od drzewa!), każda wersja musi mieć taką samą listę argumentów lub wybierać dane wejściowe w drodze analizy ogólnej listy argumentów. Każde z rozwiązań nakłada dodatkowe wymagania na kod. W rzeczywistych programach to ograniczenie może skutecznie utrudniać potencjalne zastosowania funkcji `super`, a nawet całkowicie ją wykluczać.

Aby przekonać się, że ma to znaczenie, przypomnijmy sobie klasy reprezentujące sprzedawcę pizzy z rozdziału 31. Obie klasy, zakodowane w pokazany sposób, wykorzystują bezpośrednie wywołania *po nazwie* do uruchamiania konstruktora klasy nadzędnej i automatycznie nadają argumentowi `salary` wartość zgodną z oczekiwaniami:

```
>>> class Employee:
    def __init__(self, name, salary):                      # Wspólna klasa
    nadzędna
        self.name = name
        self.salary = salary

>>> class Chef1(Employee):
    def __init__(self, name):                                # Inne argumenty
    wywołanie
        Employee.__init__(self, name, 50000)                  # Bezpośrednie

>>> class Server1(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)

>>> bob = Chef1('Robert')
>>> sue = Server1('Anna')
>>> bob.salary, sue.salary
(50000, 40000)
```

Powyższy kod działa poprawnie. Ponieważ jednak jest to drzewo pojedynczego dziedziczenia, można odczuwać pokusę wdrożenia funkcji `super` tak, aby generycznie kierowała wywołaniami konstruktora. Ten sposób działa dla każdej klasy podzędnej osobno, ponieważ lista MRO każdej z nich zawiera tylko samą klasę i jej klasę nadzelną:

```
>>> class Chef2(Employee):
    def __init__(self, name):
        super().__init__(name, 50000)                      # Wywołanie przez super

>>> class Server2(Employee):
    def __init__(self, name):
        super().__init__(name, 40000)
```

```
>>> bob = Chef2('Robert')
>>> sue = Server2('Anna')
>>> bob.salary, sue.salary
(50000, 40000)
```

Sprawdźmy jednak, co się stanie, gdy pracownik będzie należał do *obu* kategorii. Pojawią się kłopoty, ponieważ konstruktory w drzewie mają różne listy argumentów:

```
>>> class TwoJobs(Chef2, Server2): pass
>>> tom = TwoJobs('Tomasz')
TypeError: __init__() takes 2 positional arguments but 3 were given
```

Problem polega na tym, że wywołanie funkcji super w klasie Chef2 nie wywołuje już jej klasy nadzędnej Employee, tylko jej klasę bliźniaczą Server2, następną w kolejności MRO. Ponieważ ma ona inną listę argumentów niż właściwa klasa nadzędna — tylko self i name — kod ulega awarii. Jest to typowe działanie w przypadku funkcji super. Ponieważ kolejność MRO różni się w zależności od drzewa, mogą być wywoływanie różne metody. Nawet twórca klasy może nie być w stanie tego przewidzieć.

```
>>> TwoJobs.__mro__
(<class '__main__.TwoJobs'>, <class '__main__.Chef2'>, <class '__main__.Server2'>
<class '__main__.Employee'>, <class 'object'>)
>>> Chef2.__mro__
(<class '__main__.Chef2'>, <class '__main__.Employee'>, <class 'object'>)
```

Natomiast schemat bezpośrednich wywołań za pomocą nazwy działa nawet wtedy, gdy klasy są mieszane, choć uzyskane wyniki mogą być wątpliwe. Pracownik z połączonej kategorii dostaje wynagrodzenie z klasy pierwszej z lewej:

```
>>> class TwoJobs(Chef1, Server1): pass
>>> tom = TwoJobs('Tomasz')
>>> tom.salary
50000
```

Prawdopodobnie musielibyśmy skierować wywołanie do klasy z nowym wynagrodzeniem, znajdującej się na najwyższym poziomie. Ten model jest możliwy w przypadku bezpośrednich wywołań, ale nie z samą funkcją super. Co więcej, bezpośrednie wywołanie klasy Employee oznacza, że w kodzie wykorzystywane są dwie techniki kierowania wywołań, gdy w rzeczywistości wystarczy tylko jedna — bezpośrednie wywołania:

```
>>> class TwoJobs(Chef1, Server1):
    def __init__(self, name): Employee.__init__(self, name, 70000)
>>> tom = TwoJobs('Tomasz')
>>> tom.salary
70000
>>> class TwoJobs(Chef2, Server2):
    def __init__(self, name): super().__init__(name, 70000)
```

```
>>> tom = TwoJobs('Tomasz')
TypeError: __init__() takes 2 positional arguments but 3 were given
```

W tym przypadku uzasadniona byłaby generalna zmiana projektu, na przykład rozdzielenie fragmentów klas `Chef` i `Server` oraz przeniesienie ich do klas mieszanych bez konstruktora. Prawdą jest również, że polimorfizm ogólnie zakłada, że metody w interfejsie *zewnętrznym* obiektu mają takie same sygnatury argumentów. Nie dotyczy to w pełni dostosowywania metod klasy nadzędnej. Jest to *wewnętrzna* technika implementacyjna, która z natury powinna uwzględniać różnorodność metod, szczególnie w przypadku konstruktorów.

Kluczową kwestią jest jednak to, że bezpośrednie wywołania pozwalają na większą elastyczność listy argumentów, ponieważ nie uzależniają kodu od magicznej kolejności, która może różnić się w zależności od drzewa. Uogólniając, ze względu na wątpliwe (czyli ograniczone) możliwości funkcji `super` wynikające z zastępowania metod, sprzągania klas mieszanych, porządkowania wywołań i ograniczenia argumentów należy dokładnie rozważyć jej stosowanie. W miarę powiększania się drzew nawet w trybie pojedynczej instancji istotne są *późniejsze* skutki stosowania tej funkcji.

Podsumowując, źródłem większości problemów związanych ze stosowaniem funkcji `super` w tej roli są poniższe trzy wymagania:

- Metoda wywoływana przez funkcję `super` musi istnieć. Jeżeli nie ma zakotwiczenia, wymaga to napisania dodatkowego kodu.
- Metoda wywoływana przez funkcję `super` musi mieć taką samą sygnaturę argumentów w całym drzewie klas. Ogranicza to elastyczność, szczególnie metod na poziomie implementacji, na przykład konstruktorów.
- Każde oprócz ostatniego wystąpienie metody wywoływanej przez funkcję `super` musi wykorzystywać funkcję `super`. Z tego powodu trudno jest wykorzystywać istniejący kod, zmieniać kolejność wywołań, przesłaniać metody i tworzyć samodzielne klasy.

Podsumowując, jest to narzędzie dość skomplikowane i narzucające trudne kompromisy. Są to mankamenty, które ujawniają się wraz z rozwojem kodu i pojawiением wielokrotnego dziedziczenia.

Oczywiście istnieją kreatywne rozwiązania powyższych dylematów, ale konieczność wpisywania dodatkowego kodu umniejsza korzyści płynące z użycia funkcji `super`. Ponadto na ich opis nie ma tutaj miejsca. Niektóre problemy z kierowaniem metod w drzewie diamentowym można rozwiązać *bez* użycia funkcji `super`, jednak również ze względu na ograniczone miejsce pozostawię to jako ćwiczenia dla czytelnika. Ogólnie, jeżeli metody klasy nadzędnej są wywoływane jawnie za pomocą nazwy, główne klasy w drzewie diamentowym mogą sprawdzać stan w instancjach, aby uniknąć dwukrotnego wywoływanego. Jest to również skomplikowane podejście, jednak w większości przypadków rzadko wymagane, a dla niektórych programistów równie trudne jak stosowanie funkcji `super`.

Podsumowanie funkcji `super`

Tak to więc wygląda, i dobrze, i źle. Rozszerzenie to, tak jak każde w Pythonie, należy samodzielnie ocenić. Aby ułatwić podjęcie decyzji, starałem się rzetelnie przedstawić obie strony medalu. Zważywszy jednak, że:

- funkcja `super` jest inna w wersjach 2.x i 3.x,
- w wersji 3.x wykorzystuje niepythonową magię, nie można jej w pełni używać do przeciążania operatorów ani tradycyjnie kodowanych drzew wielokrotnego dziedziczenia,
- w wersji 2.x jej rola jest tak rozbudowana, że komplikuje kod, zamiast go upraszać,
- kwestionuje korzyści związane z utrzymaniem kodu, które mogą być bardziej hipotetyczne niż rzeczywiste,

nawet eksprogramiści Javy zgodzą się, że preferowana w tej książce tradycyjna technika jawnego wywoływanego klas nadzędnych za pomocą nazw jest przynajmniej tak samo skuteczna jak z użyciem funkcji `super`. W pewnych sytuacjach funkcja ta wydaje się być niezwykłą i ograniczoną odpowiedzią na pytanie, które przez większość programistów nigdy nie było zadane ani uznane za ważne w historii Pythona.

Jednocześnie wywołanie `super` oferuje jedno rozwiązywanie trudnego problemu z kierowaniem metod o takich samych nazwach w drzewach wielokrotnego dziedziczenia, pod warunkiem że stosuje się je uniwersalnie i konsekwentnie. W tym miejscu pojawia się jednak największa przeszkoda: funkcja wymaga uniwersalnego wdrażania w celu rozwiązywania problemu, z którym większość programistów prawdopodobnie się nie spotyka. Co więcej, wymaganie od programistów, aby zmienili swoje kody tak, aby użycie tego wywołania było uzasadnione, wydaje się wyjątkowo *nerealne*.

Być może głównym problemem tej roli jest *ona sama*. Wysyłanie metod o tej samej nazwie w drzewach wielokrotnego dziedziczenia jest stosunkowo rzadkie w rzeczywistych programach i na tyle niejasne, że wzbudza wiele kontrowersji i nieporozumień. Programiści nie używają Pythona w taki sam sposób jak C++, Javy czy Dylana i doświadczenia z innych języków nie zawsze się przydają.

Należy również pamiętać, że użycie funkcji `super` uzależnia program od algorytmu MRO, który został tu jedynie pobieżnie opisany ze względu na swoją złożoność. Jego zastosowanie w programach jest *sztuczne*, jest skąpo udokumentowany i słabo rozumiany w świecie Pythona. Jak widzieliśmy, jeżeli nawet pozna się algorytm MRO, jego znaczenie dla *dostosowywania, sprzągania i elastyczności kodu* są wątpliwe. Jeżeli nie zna się dokładnie tego algorytmu lub nie można przy jego użyciu osiągnąć założonych celów, lepiej nie opierać kodu na jego niejawnym wykonywaniu operacji.

Cytując fragment wyniku polecenia `import this`:

If the implementation is hard to explain, it's a bad idea.

(Jeżeli implementację trudno wytlumaczyć, to znaczy, że jest to zły pomysł).

Wywołanie `super` dobrze wpisuje się w to stwierdzenie. Większość programistów nie używa tajemniczych narzędzi do rzadkich zastosowań, bez względu na to, jak są pomysły. Dotyczy to szczególnie języka skryptowego, uważanego za przyjazny dla niespecjalistów. Niestety, programista używający jednego narzędzia może je narzuścić innym — jest to prawdziwy powód, dla którego je tutaj opisałem i wróć do niego na końcu książki.

Jak zwykle czas i baza użytkowników pokażą, czy kompromisy lub impet tego wywołania przyczynią się do jego szerszego zastosowania. Trzeba znać przynajmniej tradycyjną technikę jawnego wywoływanego klasy nadzędnej, ponieważ jest ona nadal powszechnie stosowana, często albo prostsza, albo wymagana w dzisiejszym świecie programowania w języku Python. Decydując się na stosowanie funkcji `super`, warto pamiętać, że:

- w trybie *pojedynczego dziedziczenia* funkcja może maskować późniejsze problemy i w miarę wzrostu drzewa skutkować nieoczekiwany działaniem kodu;
- w trybie *wielokrotnego dziedziczenia* funkcja znaczco komplikuje typowy przypadek użycia.

W internecie dostępne są powiązane artykuły zawierające inne opinie o funkcji `super`, jej dobrych i złych szczegółach. Można znaleźć mnóstwo opinii, jednak ostatecznie przyszłość Pythona w równym stopniu zależy od opinii czytelnika.

Rozdział 40. zawiera formalny opis pełnego dziedziczenia — procedury, której obiekty funkcji `super` unikają podczas niestandardowego kontekstowego skanowania końca kolejki MRO w poszukiwaniu pierwszego wystąpienia atrybutu (deskryptora lub wartości). Pełne dziedziczenie jest wykorzystywane w samym obiekcie `super` tylko wtedy, gdy skanowanie się nie powiedzie.



Efektem jest specjalny, rzadko spotykany przypadek podstawowego odwzorowywania nazw zarówno w języku, jak i w kodzie.

Pułapki związane z klasami

Dotarliśmy do końca opisu programowania obiektowego. W ostatniej części książki, po wyjątkach, zajmiemy się dodatkowymi przykładami i zagadnieniami dotyczącymi klas. Jednak część ta będzie w dużej mierze rozwinięciem tematów poruszonych w tym rozdziale. Jak zwykle zakończymy podrozdział standardowymi ostrzeżeniami o pułapkach, których należy unikać.

Większość problemów z klasami sprowadza się zazwyczaj do kwestii związanych z przestrzeniami nazw (co ma sens, biorąc pod uwagę to, że klasy są po prostu przestrzeniami nazw z kilkoma dodatkowymi sztuczkami). Część zagadnień omawianych w niniejszym podrozdziale to raczej studia przypadku zaawansowanego zastosowania klas niż problemy, z którymi mierzą się nawet zaawansowani programiści.

Modyfikacja atrybutów klas może mieć efekty uboczne

Teoretycznie klasy (oraz instancje klas) są obiektami *zmiennymi*. Podobnie jak wbudowane listy oraz słowniki, mogą być modyfikowane w miejscu przez przypisanie do ich atrybutów. Tak samo jak w przypadku list oraz słowników, oznacza to, że modyfikacja obiektu klasy lub instancji może mieć wpływ na liczne referencje do nich.

Zazwyczaj tego właśnie oczekujemy (w taki sposób obiekty w ogóle zmieniają swój stan), jednak świadomość tego faktu jest kwestią kluczową, kiedy modyfikuje się atrybuty klas. Ponieważ wszystkie instancje wygenerowane z klasy współdzielą przestrzeń nazw klasy, wszelkie zmiany na poziomie klasy są odzwierciedlane we wszystkich instancjach, o ile nie mają one własnych wersji zmodyfikowanych atrybutów klas.

Ponieważ klasy, moduły oraz instancje są tylko obiektami z przestrzeniami nazw atrybutów, normalnie możemy modyfikować ich atrybuty w czasie wykonywania za pomocą przypisania. Rozważmy na przykład poniższą klasę. Wewnątrz ciała klasy przypisanie do zmiennej generuje atrybut X.a, istniejący w obiekcie klasy w czasie wykonywania, który zostanie odziedziczony przez wszystkie instancje klasy X.

```
>>> class X:  
...     a = 1                                # Atrybut klasy  
...  
>>> I = X()  
>>> I.a                                    # Odziedziczyły go instancje  
1  
>>> X.a  
1
```

Jak na razie wszystko jest w porządku — sytuacja jest zupełnie normalna. Zauważmy jednak, co się stanie, kiedy zmodyfikujemy atrybut klasy w sposób dynamiczny poza instrukcją `class` — modyfikuje to atrybut także w każdej instancji dziedziczącej po tej klasie. Co więcej, nowe instancje utworzone z tej klasy podczas tej sesji czy wykonywania programu otrzymują wartość ustawioną dynamicznie, bez względu na to, co mówi kod źródłowy klasy.

```
>>> X.a = 2                                     # Może zmienić coś więcej niż
tylko X

>>> I.a                                         # I również się zmienia

2

>>> J = X( )                                    # J dziedziczy po X wartości w
czasie wykonywania

>>> J.a                                         # (przypisanie do J.a zmienia a
w J, a nie w X lub I)

2
```

Czy jest to przydatna opcja, czy też niebezpieczna pułapka? Sami musimy to ocenić. Jak mieliśmy okazję przekonać się w rozdziale 27., możemy tak naprawdę wykonać swoją pracę, modyfikując atrybuty klasy i nie tworząc nigdy żadnej instancji. Technika ta może symulować „rekordy” lub „struktury” z innych języków programowania. Jako powtórkę rozważmy poniższy, dość niezwykły, ale całkowicie poprawny program.

```
class X: pass                                     # Utworzenie kilku przestrzeni
nazw atrybutów

class Y: pass

X.a = 1                                           # Wykorzystanie atrybutów klas
jako zmiennych

X.b = 2                                           # Nigdzie nie ma instancji

X.c = 3

Y.a = X.a + X.b + X.c

for X.i in range(Y.a): print X.i                # Wyświetla 0..5
```

W powyższym kodzie klasy X oraz Y działają jak moduły bez plików — przestrzenie nazw służące do przechowywania zmiennych, których konfliktu wolelibyśmy uniknąć. Jest to całkowicie poprawna sztuczka programistyczna w Pythonie, która jednak jest mniej właściwa, kiedy stosuje się ją do klas utworzonych przez inne osoby. Nie zawsze możemy być pewni, że modyfikowane atrybuty klas nie są krytycznym elementem wewnętrznego zachowania klasy. Jeśli chcemy symulować struktury z języka C, być może lepiej będzie modyfikować instancje niż klasy, ponieważ w ten sposób ma to wpływ jedynie na jeden obiekt.

```
class Record: pass

X = Record( )

X.name = 'robert'

X.job = 'Twórca pizzy'
```

Modyfikowanie mutowalnych atrybutów klas również może mieć efekty uboczne

Ta pułapka to w zasadzie rozbudowana wersja poprzedniej. Atrybuty klas są współdzielone przez wszystkie instancje. Jeśli atrybut klasy zawiera referencję do obiektu mutowalnego, modyfikacja tego obiektu z poziomu instancji będzie miała wpływ na wszystkie instancje klasy.

```
>>> class C:  
...     shared = []                      # Atrybut klasy  
...     def __init__(self):  
...         self.perobj = []                # Atrybut instancji  
...  
>>> x = C()                          # Dwie instancje  
>>> y = C()                          # Atrybuty klasy są współdzielone  
>>> y.shared, y.perobj  
([], [])  
>>> x.shared.append('spam')          # Również modyfikuje zachowanie instancji  
y!  
>>> x.perobj.append('spam')          # Modyfikuje zachowanie instancji x  
>>> x.shared, x.perobj  
(['spam'], ['spam'])  
>>> y.shared, y.perobj             # y widzi zmiany wprowadzone w x  
(['spam'], [])  
>>> C.shared                        # Zapisane w klasie i współdzielone  
['spam']
```

Ten efekt nie różni się zbytnio od zjawisk, które poznaliśmy w tej książce: obiekty mutowalne są współdzielone przez zwykłe zmienne, zmienne globalne są współdzielone przez funkcje, obiekty zdefiniowane w modułach są współdzielone przez moduły je importujące, mutowalne argumenty funkcji są współdzielone przez kod wywołujący i kod wywoływany. Wszystko to są przykłady wielokrotnych referencji do obiektów mutowalnych i wszystkie one powodują efekty uboczne, jeśli obiekt mutowalny zostanie zmodyfikowany w jednej z referencji. W tym przypadku mówimy o atrybutach klas dziedziczących przez instancje, ale nadal chodzi o to samo zjawisko. Efekt będzie zależny od tego, czy modyfikowany jest obiekt wskazany przez referencję, czy sama referencja:

```
x.shared.append('spam')  # Modyfikuje współdzielony obiekt przypisany  
atrybutowi klasy  
x.shared = 'spam'        # Modyfikuje lub tworzy nowy atrybut instancji przez  
przypisanie
```

Należy pamiętać, że omawiane efekty uboczne nie wynikają z błędów w implementacji języka. Są to subtelności jego semantyki, których należy mieć świadomość, ale które mogą mieć zupełnie legalne zastosowania w programach w Pythonie.

Dziedziczenie wielokrotne – kolejność ma znaczenie

Może to być oczywiste, ale warto to podkreślić jeszcze raz. Jeśli korzystamy z dziedziczenia wielokrotnego, kolejność podawania klas nadzujących w nagłówku instrukcji `class` może być

kwestią kluczową. Python zawsze przeszukuje klasy nadrzędne od lewej do prawej strony, zgodnie z ich kolejnością w wierszu nagłówka.

Weźmy przykład wielokrotnego dziedziczenia przedstawiony w rozdziale 31. Założymy, że dodatkowo zaimplementowaliśmy metodę `__str__` w klasie Super.

```
class ListTree:  
    def __str__(self): ...  
  
class Super:  
    def __str__(self): ...  
  
class Sub(ListTree, Super):      # Wymuszamy użycie metody __str__ z klasy  
    ListTree  
  
x = Sub()                      # Mechanizm dziedziczenia znajdzie metodę w  
    ListTree zanim przeszuka Super
```

Z której klasy będziemy dziedziczyć: `ListTree` czy `Super`? Wyszukiwanie nazw odbywa się od lewej do prawej, zatem używana będzie metoda tej klasy, która jest zadeklarowana wcześniej (bardziej po lewej) w nagłówku klasy `Sub`. Założymy, że chcemy użyć metody `__str__` z klasy `ListTree`, dlatego zadeklarujemy ją jako pierwszą (w przedstawionym w rozdziale 31. przykładzie klasa `tkinter.Button` miała zdefiniowaną własną metodę `__str__`).

Założymy jednak, że `Super` i `ListTree` posiadają inne atrybuty o takich samych nazwach, których chcielibyśmy użyć. Jeśli chcemy użyć jednej nazwy z `Super`, a innej z `ListTree`, nie pomoże zmiana kolejności deklaracji klas nadrzędnych w klasie `Sub`. Możemy jednak ręcznie zadeklarować każdy z tych atrybutów w klasie `Sub`:

```
class ListTree:  
    def __str__(self): ...  
  
    def other(self): ...  
  
class Super:  
    def __str__(self): ...  
  
    def other(self): ...  
  
class Sub(ListTree, Super):      # Wymuszamy użycie metody __str__ z klasy  
    ListTree  
  
        other = Super.other          # ale metodę other wybieramy z klasy Super  
  
    def __init__(self):  
        ...  
  
x = Sub()                      # Mechanizm dziedziczenia przeszuka klasę Sub  
    przed ListTree/Super
```

W powyższym kodzie przypisanie do zmiennej `other` wewnętrz klasy `Sub` tworzy `Sub.other` — referencję z powrotem do obiektu `Super.other`. Ponieważ zmienna ta znajduje się niżej w drzewie, `Sub.other` w rezultacie ukrywa `ListTree.other`, czyli atrybut, który znalazłyby normalne wyszukiwanie dziedziczenia. I podobnie, gdybyśmy podali `Super` jako pierwszą w nagłówku klasy, tak by wybrać atrybut `other` z tej właśnie klasy, musielibyśmy ręcznie wybrać metodę z klasy `ListTree`.

```
class Sub(Super, Lister):          # Pobranie other z klasy Super  
    dzięki kolejności
```

```
__repr__ = Lister.__repr__ # Jawný wybór Lister.__repr__
```

Dziedziczenie wielokrotne jest narzędziem zaawansowanym. Nawet jeśli rozumiemy poprzedni akapit, lepiej jest korzystać z dziedziczenia wielokrotnego oszczędnie i ostrożnie. W przeciwnym razie znaczenie nazwy w naszym kodzie może zależeć od kolejności, w jakiej klasy są mieszane w dowolnie odległej klasie podzielonej. Inny przykład zaprezentowanej tu techniki wymuszania atrybutów można znaleźć w omówieniu jawnego rozwiązywania konfliktów w podrozdziale „Klasy w nowym stylu” i w opisie funkcji `super` zamieszczonym wcześniej w niniejszym rozdziale.

Z reguły dziedziczenie wielokrotne działa najlepiej, kiedy nasze klasy mieszane są na tyle samodzielne, na ile jest to możliwe. Ponieważ mogą być wykorzystywane w wielu różnych kontekstach, nie powinny czynić żadnych założeń dotyczących zmiennych znajdujących się w innych klasach drzewa. Omówiona w rozdziale 31. opcja atrybutów pseudoprivaćnych `_X` może pomóc uczynić nazwy, na których klasa polega, bardziej lokalnymi, a także ograniczyć zmienne dodawane przez klasy mieszane. W tym przykładzie, jeśli na przykład klasa `ListTree` chce eksportować jedynie własną metodę `__str__`, drugą swoją metodę może nazwać `__other`, by uniknąć konfliktów z innymi klasami.

Zakresy w metodach i klasach

Analizując znaczenie nazw w kodzie opartym na klasach, należy pamiętać, że klasy wprowadzają zakresy lokalne, podobnie jak funkcje, a metody są funkcjami jeszcze bardziej zagnieżdżonymi. W poniższym przykładzie funkcja `generate` zwraca instancję zagnieżdżonej klasy `Spam`. Wewnątrz jej kodu nazwa klasy `Spam` przypisywana jest w zakresie lokalnym funkcji `generate`. Jest więc widoczna dla wszystkich zagnieżdżonych funkcji, włącznie z kodem wewnątrz metody `method`. Odpowiada to literze E w regule LEGB przeszukiwania zakresów widoczności.

```
def generate():
    class Spam: # Spam jest nazwą zawartą w lokalnym
        zakresie metody generate
            count = 1
            def method(self):
                print(Spam.count) # Nazwa widoczna w zakresie metody generate
                zgodnie z regułą LEGB (litera E)
            return Spam()
    generate().method()
```

Przykład ten działa w Pythonie 2.2 oraz nowszych wersjach, ponieważ zakresy lokalne wszystkich instrukcji `def` zawierających funkcję są automatycznie widoczne dla `def` zagnieżdżonych (w tym zagnieżdżonych `def` metod, jak w powyższym kodzie).

Warto zauważyć, że instrukcje `def` nie widzą zakresu lokalnego klasy zawierającej — widzą jedynie zakresy lokalne zawierających instrukcję `def`. Z tego powodu metody muszą przechodzić przez instancję `self` lub nazwę klasy, by móc się odnieść do metod oraz innych atrybutów zdefiniowanych w obejmującej je instrukcji `class`. Kod metody musiał na przykład używać `self.count` lub `Spam.count` zamiast po prostu `count`.

Aby uniknąć zagnieżdżenia, można zmienić kod tak, aby klasa `Spam` była zdefiniowana na najwyższym poziomie modułu. Wtedy zarówno zagnieżdżona funkcja `method`, jak i metoda `generate` z najwyższego poziomu znajdą klasę `Spam` w swoich globalnych zakresach. Klasa nie będzie zlokalizowana w zakresie funkcji, ale będzie lokalna w zakresie pojedynczego modułu:

```

def generate():
    return Spam()

class Spam:                                # Zdefiniowana na najwyższym
    poziomie modułu

    count = 1

    def method(self):
        print(Spam.count)                  # Działa: w zakresie globalnym
        (modułu)

    generate().method()

```

Tak naprawdę takie rozwiązanie jest zalecane we wszystkich wydaniach Pythona — kod jest prostszy, jeśli unikamy zagnieżdżania klas oraz funkcji. Z drugiej strony zagnieżdżanie klas jest przydatne w *kontekście domknięć*, w których zakres otaczającej funkcji zachowuje *stan* wykorzystywany przez klasę lub jej metody. W poniższym przykładzie zagnieżdżona metoda `method` ma dostęp do zakresu własnego, zakresu otaczającej ją funkcji (argumentu `label`), globalnego zakresu otaczającego ją modułu, wszystkich danych zapisanych przez klasę w instancji `self` i samej klasy poprzez jej nielokalną nazwę:

```

>>> def generate(label):      # Zwraca klasę, a nie instancję
    class Spam:
        count = 1
        def method(self):
            print("%s=%s" % (label, Spam.count))
        return Spam

>>> aclass = generate('Gotchas')
>>> I = aclass()
>>> I.method()
Gotchas=1

```

Różne pułapki związane z klasami

Poniżej przedstawionych jest kilka ostrzeżeń, w większości dla przypomnienia.

Rozsądnie wybieraj miejsce przechowywania atrybutu w instancji lub w klasie

Ostrożnie decyduj, czy atrybut ma być przechowywany w klasie, czy w jej instancjach. W pierwszym przypadku miejsce będzie współdzielone przez wszystkie instancje, a w drugim w każdej instancji będzie inne. Ten wybór może mieć w praktyce krytyczne znaczenie. Jeżeli na przykład w programie z interfejsem graficznym pewna informacja (np. ostatni katalog użyty podczas zapisywania pliku lub wprowadzone wcześniej hasło) ma być współdzielona przez wszystkie tworzone przez aplikację obiekty klasy okna, musi być zapisywana na poziomie klasy. Gdyby była zapisywana w instancji jako atrybut `self`, wtedy podczas przeszukiwania drzewa dziedziczenia byłaby dla każdego okna inna lub nie byłoby jej w ogóle.

Zazwyczaj wywołuj konstruktory klasy nadzędnej

Jak pamiętam, Python podczas tworzenia instancji uruchamia tylko jedną metodę `__init__` konstruktora, znajdująca się najniżej w drzewie dziedziczenia klas. Nie uruchamia automatycznie konstruktorów wszystkich klas nadzędnych. Ponieważ konstruktory wykonują wymagane operacje inicjujące, zazwyczaj same powinny wywoływać konstruktor klasy nadzędnej (jawnie wykorzystując nazwę klasy nadzędnej lub funkcję `super` i przekazując wszystkie wymagane argumenty), chyba że całkowicie zastępują konstruktor klasy nadzędnej lub klasa nadzędna nie ma lub nie dziedziczy konstruktora w ogóle.

Klasy wykorzystujące delegację w 3.x — `_getattr_` i funkcje wbudowane

Dla przypomnienia: jak pisałem wcześniej w tym rozdziale i w innych miejscach książki, klasy wykorzystujące metodę przeciążającą operator `__getattr__` w celu delegowania operacji pobierania atrybutu do obiektu opakowującego nie będą działać w 3.x (i w 2.x w przypadku użycia klas w nowym stylu) chyba że przeciążenie operatora zostanie zdefiniowane również w klasie opakowującej. Aby rozwiązać ten problem metody specjalne muszą być zdefiniowane w klasach opakowujących ręcznie, za pomocą narzędzi pomocniczych lub przez definicje w klasach nadzędnych. W rozdziale 40. dowiemy się, jak to zrobić.

Przesadne opakowywanie

Kiedy się je dobrze stosuje, możliwości ponownego wykorzystania kodu w programowaniu zorientowanym obiektowo potrafią znacznie skrócić czas programowania. Czasami jednak potencjału abstrakcyjnego programowania zorientowanego obiektowo można nadużyć, tak że kod staje się trudny do zrozumienia. Jeżeli zależności między klasami są zbyt głębokie, wtedy kod staje się nieczytelny i aby zrozumieć, co robi jakaś operacja, czasami konieczne staje się przeszukanie wielu klas.

Pracowałem kiedyś w firmie programującej w języku C++, która stosowała tysiące klas (część z nich była generowana automatycznie) i do piętnastu poziomów dziedziczenia. Rozszerzanie wywołań metod w tak skomplikowanych systemach było często monumentalnym zadaniem — nawet dla najbardziej podstawowych operacji konieczne było sprawdzanie wielu klas. Logika systemu była tak naprawdę tak głęboko opakowana, że zrozumienie fragmentu kodu wymagało w niektórych przypadkach całych dni przekopywania się przez powiązane z nim pliki. Z punktu widzenia produktywności programisty na pewno nie jest to idealna sytuacja!

Tutaj również ma zastosowanie najważniejsza reguła programowania w Pythonie — *nie należy czegoś komplikować, o ile nie jest to naprawdę konieczne*. Opakowanie kodu w większą liczbę warstw klas w taki sposób, by stał się on niezrozumiały, zawsze jest złym pomysłem. Abstrakcja jest podstawą polimorfizmu oraz hermetyzacji i może być bardzo skutecznym narzędziem, kiedy się ją dobrze wykorzystuje. Ułatwimy sobie jednak debugowanie i utrzymywanie kodu, jeśli interfejsy naszych klas będą intuicyjne, będziemy unikać nadmiernej abstrakcji kodu i utrzymywać hierarchie klas krótkie oraz płaskie, o ile nie mamy istotnego powodu, by robić inaczej. Należy pamiętać, że tworzony przez nas kod zazwyczaj będą musieli czytać inni programiści. Więcej informacji na temat zasady KISS zawiera rozdział 20.

Podsumowanie rozdziału

Niniejszy rozdział przedstawił kilka zaawansowanych zagadnień związanych z klasami, w tym tworzenie klas podzędnych typów wbudowanych, klasy w nowym stylu, metody statyczne oraz dekoratory funkcji. Większość tych elementów to opcjonalne rozszerzenia modelu programowania zorientowanego obiektowo w Pythonie, jednak mogą one stać się bardziej użyteczne w miarę pisania większych programów zorientowanych obiektowo. Jak wspomniałem wcześniej, nasza dyskusja dotycząca bardziej zaawansowanych narzędzi programowania

obiektowego będzie kontynuowana w ostatniej części książki. Tam znajdują się omówienia zaawansowanych technik, jak właściwości, deskryptory, dekoratory i metaklasy.

Jest to koniec części książki poświęconej klasom, dlatego na końcu rozdziału znajduje się zbiór ćwiczeń praktycznych — należy koniecznie się z nimi zapoznać w celu zdobycia doświadczenia w pisaniu kodu prawdziwych klas. W kolejnym rozdziale zaczniemy przyglądać się ostatniemu podstawowemu elementowi języka, czyli *wyjątkom*. Wyjątki są w Pythonie mechanizmem służącym do komunikowania błędów oraz innych warunków kodu. Jest to stosunkowo lekki temat, jednak zachowałem go na koniec, ponieważ wyjątki powinny obecnie być zapisywane jako klasy. Zanim jednak przejdziemy do tego zagadnienia, warto zająć się quizem podsumowującym niniejszy rozdział oraz ćwiczeniami kończącymi tę część książki.

Sprawdź swoją wiedzę — quiz

1. Należy podać dwa sposoby rozszerzania wbudowanego typu obiektu.
2. Do czego wykorzystywane są dekoratory funkcji i klas?
3. W jaki sposób tworzy się kod klas w nowym stylu?
4. Czym różnią się od siebie klasy z modelu klasycznego oraz klasy w nowym stylu?
5. Czym różnią się od siebie metody normalne i statyczne?
6. Czy stosowanie w kodzie narzędzi takich jak `__slots__` i `super` jest właściwe?
7. Do ilu powinniśmy zliczyć przed rzuceniem Świętego Granatu Ręcznego?

Sprawdź swoją wiedzę — odpowiedzi

1. Możemy osadzić wbudowany obiekt w klasie opakowującej lub bezpośrednio utworzyć jego klasę podrzędną. To drugie rozwiązanie zazwyczaj jest prostsze, gdyż automatycznie dziedziczona jest większość oryginalnego zachowania.
2. Dekoratory funkcji są zazwyczaj wykorzystywane w celu dodania do istniejącej funkcji nowej warstwy logiki, która jest wykonywana za każdym wywołaniem funkcji. Mogą być używane do logowania lub zliczania wywołań funkcji czy sprawdzania jej typów argumentów. Są również wykorzystywane do „deklarowania” metod statycznych (prostych funkcji w klasach, do których nie przekazuje się instancji), metod i właściwości. Dekoratory klas działają podobnie, jednak zarządzają całymi obiekttami i ich interfejsami, a nie wywołaniami funkcji.
3. Klasy w nowym stylu tworzy się poprzez dziedziczenie po wbudowanej klasie `object` (lub dowolnym innym typie wbudowanym). W Pythonie 3.x wszystkie klasy są tworzone w nowym stylu, więc takie dziedziczenie nie jest wymagane (niemniej jest dopuszczalne). W 2.x klasy jawnie dziedziczące po typie wbudowanym lub klasie `object` są klasami w nowym stylu, pozostałe są klasami „klasycznymi”.
4. Klasy w nowym stylu inaczej przeszukują drzewa z wielokrotnym dziedziczeniem po jednej klasie nadrzędnej (model diamentowy) — najpierw szukają na szerokość zamiast na głębokość (w góre). Klasy w nowym stylu zwracają inne wyniki funkcji wbudowanej `type` dla instancji i klas i nie wykorzystują uogólnionych metod pobierania atrybutów (jak `__getattr__`) dla operacji wbudowanych, obsługując

również zbiór dodatkowych, zaawansowanych narzędzi, w tym właściwości oraz listę atrybutów instancji `__slots__`.

5. Zwykłe metody (instancji) otrzymują argument `self` (domniemaną instancję), natomiast w przypadku metod statycznych tak nie jest. Metody statyczne są prostymi funkcjami zagnieźdzonymi w obiekcie klasy. By metoda stała się statyczna, trzeba ją przekształcić za pomocą specjalnej funkcji wbudowanej lub użyć składni dekoratora. W Pythonie 3.x proste funkcje klasy można wywoływać z pominięciem powyższego kroku, natomiast wywołanie za pomocą instancji wciąż wymaga użycia deklaracji statycznej metody.
6. Oczywiście tak, jednak nie należy automatycznie stosować zaawansowanych narzędzi bez dokładnego przeanalizowania wywoływanych przez nie skutków. Na przykład sloty mogą zakłócić działanie kodu, funkcja `super` użyta w drzewie pojedynczego dziedziczenia może maskować późniejsze problemy, a w drzewie wielokrotnego dziedziczenia znacznie komplikować kod w izolowanym przypadku użycia. Aby była najbardziej użyteczna w obu sytuacjach, wymagane jest jej uniwersalne wdrożenie. Ocena nowych lub zaawansowanych narzędzi jest podstawowym zadaniem każdego inżyniera, dlatego w tym rozdziale tak dokładnie opisane są wszystkie kompromisy. Celem tej książki nie jest nauczenie programisty, którego narzędzia ma używać, tylko podkreślenie znaczenia ich obiektywnej analizy. W dziedzinie programowania to zadanie często ma zbyt niski priorytet.
7. Trzy sekundy. A dokładniej: „I Pan powiedział: — Wpierw wyjąć musisz Świętą Zawleczkę. Potem masz zliczyć do trzech, nie mniej, nie więcej. Trzy ma być liczbą, do której liczyć masz, i liczbą tą ma być trzy. Do czterech nie wolno ci liczyć ani do dwóch. Masz tylko policzyć do trzech. Pięć jest wykluczone. Gdy liczba trzy jako trzecia w kolejności osiągnięta zostanie, wówczas rzucić masz Święty Granat Ręczny z Antiochii w kierunku wroga, co naigrywał się z ciebie w polu widzenia twoego, a on kitę odwali”^[4].

Sprawdź swoją wiedzę – ćwiczenia do części szóstej

Poniższe ćwiczenia będą wymagały samodzielnego napisania kilku klas oraz poeksperymentowania z istniejącym kodem. Problem istniejącego kodu polega oczywiście na tym, że musi on istnieć. By pracować z klasą zbioru z ćwiczenia 5., można albo pobrać kod źródłowy z internetu (zgodnie z informacjami z Przedmowy), albo wpisać go ręcznie (jest on stosunkowo krótki). Programy te zaczynają być coraz bardziej zaawansowane, dlatego należy koniecznie sprawdzić rozwiązania znajdujące się na końcu książki w celu odnalezienia wskazówek. Można je znaleźć w podrozdziale „Część VI. Klasy i programowanie zorientowane obiektowo” w dodatku D.

1. *Dziedziczenie.* Należy napisać klasę o nazwie `Adder`, która eksportuje metodę `add(self, x, y)` wyświetlającą komunikat: „Nie zaimplementowano”. Następnie należy zdefiniować dla niej dwie klasy podzielne implementujące metodę `add`:

`ListAdder`

Z metodą `add` zwracającą konkatenację dwóch argumentów będących listami.

`DictAdder`

Z metodą `add` zwracającą nowy słownik zawierający elementy znajdujące się w obydwu słownikach argumentów (wystarczy dowolna definicja dodawania).

Należy poeksperymentować, tworząc instancje wszystkich trzech klas w sesji interaktywnej i wywołując ich metody `add`.

Następnie należy rozszerzyć klasę nadziedzną `Adder`, tak by zapisywała obiekt w instancji za pomocą konstruktora (na przykład przypisując do `self.data` listę lub słownik) i przeciążała operator `+` za pomocą metody `__add__`, tak by automatycznie przesyłać argumenty do metod `add` (na przykład `X + Y` ma wywoływać `X.add(X.data, Y)`). Jakie jest najlepsze miejsce na umieszczenie konstruktorów oraz metod przeciążania operatorów (to znaczy w których klasach powinny się one znaleźć)? Jakie rodzaje obiektów można dodać do instancji klas?

W praktyce łatwiejsze może okazać się zapisanie kodu metod `add` w taki sposób, by przyjmowały one tylko jeden prawdziwy argument (na przykład `add(self, y)`) i dodawały ten argument do bieżących danych instancji (na przykład `self.data + y`). Czy ma to większy sens od przekazania dwóch argumentów do metody `add`? Czy klasy stają się w ten sposób bardziej zorientowane obiektywem?

2. *Przeciążanie operatorów.* Należy napisać klasę `MyList` opakowującą listę Pythona. Powinna ona przeciągać większość operatorów oraz operacji listy, w tym `+`, indeksowanie, iterację, wycinki oraz metody listy, takie jak `append` oraz `sort`. Listę wszystkich możliwych metod, jakie można obsługiwać, można znaleźć w dokumentacji Pythona. Należy również udostępnić konstruktor dla tej klasy, który tworzy istniejącą listę (lub instancję `Mylist`) i kopiuje jej komponenty do składowej instancji. Należy poeksperymentować z tą klasą w sesji interaktywnej. Rzeczy, które należy sprawdzić:
 - a. Dlaczego skopiowanie początkowej wartości jest tutaj tak istotne?
 - b. Czy można wykorzystać pusty wycinek (na przykład `start[:]`) do skopiowania początkowej wartości, jeśli jest to instancja klasy `MyList`?
 - c. Czy istnieje jakiś uniwersalny sposób przekierowania wywołań metod listy do opakowanej listy?
 - d. Czy można dodać do siebie instancję klasy `MyList` oraz zwykłą listę? A co w przypadku zwykłej listy i instancji `MyList`?
 - e. Obiekt jakiego typu powinny zwracać operacje takie, jak `+` czy wycinki? Co z operacjami indeksowania?
 - f. Jeśli pracujemy w którejś z nowszych wersji Pythona (od 2.2 w góre), możemy zaimplementować ten typ klasy opakowującej, osadzając prawdziwą listę w samodzielnej klasie lub rozszerzając wbudowany typ listy za pomocą klasy podrzędnej. Które rozwiązanie jest prostsze i dlaczego?
3. *Klasy podrzędne.* Należy utworzyć klasę podrziedzną dla `MyList` z ćwiczenia 2. o nazwie `MyListSub`. Klasa ta powinna rozszerzać `MyList` i wyświetlać komunikat do `stdout` przed wywołaniem każdej przeciążonej operacji oraz zliczać wywołania. Klasa `MyListSub` powinna dziedziczyć podstawowe metody zachowania po `MyList`. Dodanie sekwencji do `MyListSub` powinno wyświetlić komunikat, dokonać inkrementacji licznika wywołań `+` i wykonać metodę klasy nadziedznej. Należy również dodać nową metodę wyświetlającą liczniki operacji do `stdout` i poeksperymentować z tą klasą w sesji interaktywnej. Czy liczniki zliczają wywołania na instancję, czy też na klasę (dla wszystkich instancji klasy)? W jaki sposób można w kodzie zaimplementować obie wersje? Wskazówka: jest to uzależnione od tego, do

którego obiektu przypisane są składowe licznika — składowe klasy są współdzielone przez instancje, natomiast składowe `self` są danymi poszczególnych instancji.

4. *Metody atrybutów.* Należy napisać klasę o nazwie `Attrs` z metodami przechwytyującymi każdą kwalifikację atrybutów (zarówno pobrania, jak i przypisania) i wyświetlającymi do `stdout` komunikaty wymieniające argumenty. Później trzeba utworzyć instancję klasy `Attrs` i poeksperymentować ze składnią kwalifikującą w sesji interaktywnej. Co się dzieje, jeśli próbujemy wykorzystać instancję w wyrażeniach? Należy spróbować dodać, zindeksować i wykonać wycinek klasy. (Uwaga: w pełni uogólnione podejście wykorzystujące metodę `__getattr__` zadziała w zwykłej klasie w wersji 2.x, ale nie w klasie w nowym stylu w wersji 3.x — z powodów opisanych w rozdziałach 28., 31. i 32. i przytoczonych w rozwiążaniu tego ćwiczenia).
5. *Obiekty zbiorów.* Należy poeksperymentować z klasą zbioru opisaną w podrozdziale „Rozszerzanie typów za pomocą osadzania”. Należy wykonać polecenia pozwalające na następujące rodzaje operacji:
 - a. Należy utworzyć dwa zbiory liczb całkowitych i obliczyć ich część wspólną oraz sumę za pomocą wyrażeń z operatorami & oraz |.
 - b. Należy utworzyć zbiór z łańcucha znaków i poeksperymentować z indeksowaniem tego zbioru. Jakie metody klasy są wywoływane?
 - c. Należy za pomocą pętli `for` spróbować wykonać iterację po elementach zbioru łańcucha znaków. Jakie metody wykonywane są tym razem?
 - d. Należy spróbować obliczyć część wspólną oraz sumę naszego zbioru łańcucha znaków oraz prostego łańcucha znaków. Czy to zadziała?
 - e. Teraz należy rozszerzyć zbiór, tworząc klasę podrzędną obsługującą dowolną liczbę argumentów, wykorzystując do tego formę `*args`. Wskazówka: wersja tych algorytmów przeznaczona dla funkcji znajduje się w rozdziale 18. Należy obliczyć części wspólne oraz sumy większej liczby argumentów za pomocą naszej klasy podrzędnej. W jaki sposób można uzyskać część wspólną trzech lub większej liczby zbiorów, biorąc pod uwagę to, że operator & ma tylko dwie strony?
 - f. W jaki sposób można emulować inne operacje na listach w klasie zbioru? Wskazówka: metoda `__add__` może przechwytywać konkatenację, a `__getattr__` odwołania wykorzystywane w większości metod, np. `append`.
6. *Łącza drzew klas.* W podrozdziałach „Przestrzenie nazw — cała historia” w rozdziale 29. oraz „Dziedziczenie wielokrotne — klasy mieszane” w rozdziale 31. wspomniałem, że klasy mają atrybut `__bases__` zwracający krotkę obiektów ich klas nadrzędnych (tych wymienionych w nawiasach w nagłówku klasy). Należy wykorzystać atrybut `__bases__` do rozszerzenia klas zdefiniowanych w module `lister.py` (rozdział 31.) w taki sposób, by wyświetlała ona nazwy bezpośrednich klas nadrzędnych klasy instancji. W rezultacie pierwszy wiersz reprezentacji łańcuchów znaków powinien wyglądać jak poniższy (adres może być inny):
`<Instancja klasy Sub(Super, Lister), adres 7841200:`
7. *Kompozycja.* Należy wykonać symulację scenariusza zamawiania jedzenia w barze szybkiej obsługi, definiując cztery klasy:
 - Lunch
 - Pojemnik oraz klasa kontrolera.

Customer

Aktor kupujący jedzenie.

Employee

Aktor, u którego klient składa zamówienie.

Food

To, co kupuje klient.

Na początek w poniższym kodzie widoczne są klasy oraz metody, jakie będziemy definiować.

```
class Lunch:  
    def __init__(self)                      # Tworzy/osadza klasy  
        Customer oraz Employee  
  
    def order(self, foodName)                # Rozpoczęcie  
        symulacji zamówienia klienta Customer  
  
    def result(self)                       # Zapytanie klienta  
        Customer, jakie ma jedzenie Food  
  
class Customer:  
    def __init__(self)                      # Inicjalizacja  
        mojego jedzenia na None  
  
    def placeOrder(self, foodName, employee) # Złożenie  
        zamówienia pracownikowi Employee  
  
    def printFood(self)                    # Wyświetlenie nazwy  
        mojego jedzenia  
  
class Employee:  
    def takeOrder(self, foodName)          # Zwraca jedzenie  
        Food z żądaną nazwą  
  
class Food:  
    def __init__(self, name)               # Przechowanie nazwy  
        jedzenia
```

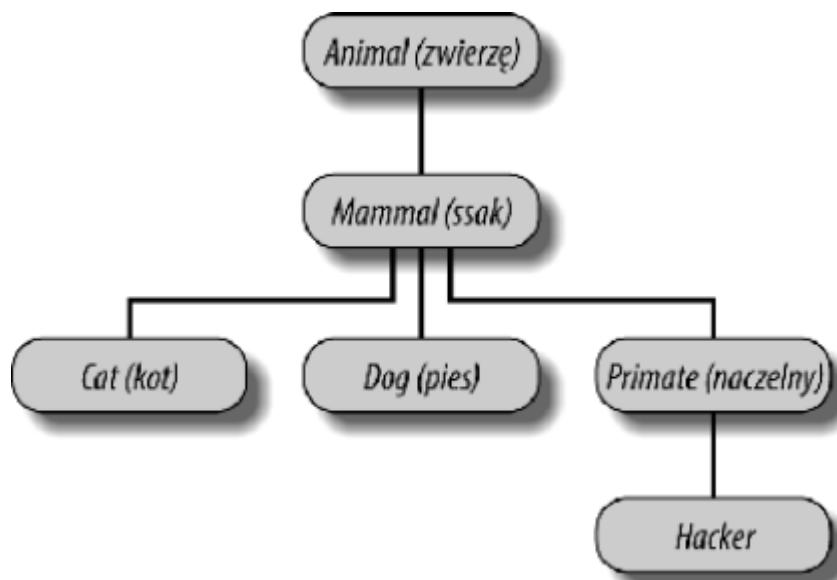
Symulacja zamówienia przebiega następująco:

- a. Konstruktor klasy `Lunch` powinien utworzyć oraz osadzić instancję klas `Customer` (klient) oraz `Employee` (pracownik). Powinien również eksportować metodę o nazwie `order`. Po wywołaniu metoda `order` powinna poprosić klienta `Customer` o złożenie zamówienia, wywołując jego metodę `placeOrder`. Metoda `placeOrder` klasy `Customer` powinna z kolei poprosić obiekt pracownika `Employee` o nowy obiekt jedzenia `Food`, wywołując jego metodę `takeOrder`.
- b. Obiekty jedzenia `Food` powinny przechowywać łańcuch znaków z nazwą jedzenia (na przykład „burrito”), przekazany z `Lunch.order` do `Customer.placeOrder`, stamtąd do `Employee.takeOrder`, a na końcu do konstruktora obiektu `Food`. Klasa najwyższego poziomu `Lunch` powinna również eksportować metodę o nazwie `result`, prosiącą klienta o wyświetlenie nazwy jedzenia, jakie otrzymał od pracownika za pomocą zamówienia (można wykorzystać to do przetestowania naszej symulacji).

Należy zauważyć, że aby klasa `Customer` mogła wywoływać metody klasy `Employee`, klasa `Lunch` musi przekazać klasę `Employee` lub samą siebie do klasy `Customer`.

Należy poeksperymentować z tymi klasami w sesji interaktywnej, importując klasę `Lunch`, wywołując jej metodę `order` w celu wykonania interakcji, a następnie wywołując jej metodę `result` w celu zweryfikowania, że klient `Customer` otrzymał to, co zamówił. Jeśli wolimy, możemy również zapisać przypadki testowe jako kod samosprawdzający w pliku, w którym klasy zostały zdefiniowane, wykorzystując do tego sztuczkę z nazwą `__name__` modułu z rozdziału 25. W tej symulacji to klasa `Customer` jest aktywnym agentem. W jaki sposób zmieniłyby się klasy, gdyby to `Employee` był obiektem inicjalizującym interakcję pomiędzy klientem a pracownikiem?

8. *Hierarchia zwierząt w zoo.* Należy rozważyć drzewo klas pokazane na rysunku 32.1.



Rysunek 32.1. Hierarchia zwierząt w zoo składająca się z klas połączonych w drzewo, które jest przeszukiwane przez dziedziczenie atrybutów. Zwierzęta mają wspólną metodę `reply`, jednak każda klasa ma własną metodę `speak` wywoływaną przez `reply`

Tym razem trzeba utworzyć sześć instrukcji `class` pozwalających utworzyć model tej taksonomii za pomocą mechanizmu *dziedziczenia* Pythona. Następnie należy dodać metodę `speak` do każdej klasy, wyświetlającą unikalny komunikat, oraz metodę `reply` w klasie nadrzędnej najwyższego poziomu `Animal`, która wywołuje po prostu `self.speak` w celu wywołania komunikatu specyficznego dla tej kategorii w klasie podrzędnej znajdującej się niżej (pozwoli to uruchomić wyszukiwanie dziedziczenia niezależne od `self`). Na koniec należy usunąć metodę `self` z klasy `Hacker`, tak by przyjęła ona metodę domyślną klasy nadrzędnej. W rezultacie klasy powinny działać w następujący sposób.

```
% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
```

```

>>> spot.reply()                                # Animal.reply;
wywołuje Cat.speak

miau

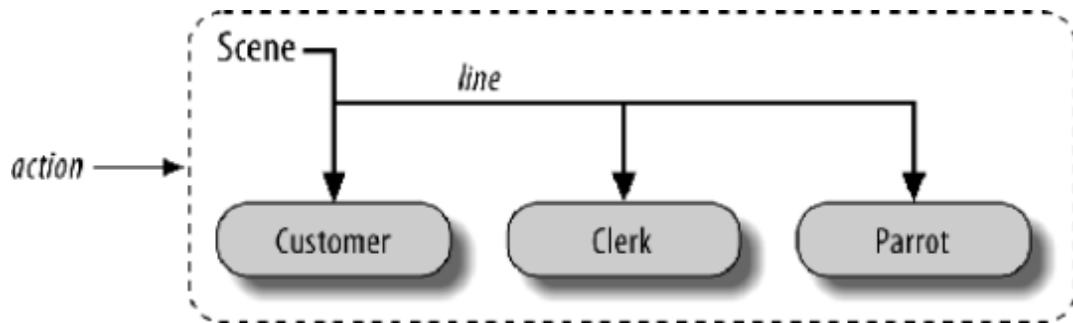
>>> data = Hacker()                           # Animal.reply;
wywołuje Primate.speak

>>> data.reply()

Witaj, świecie!

```

9. *Skecz z martwą papugą.* Rozważmy strukturę osadzania obiektów przedstawioną na rysunku 32.2.



Rysunek 32.2. Kompozyt sceny z klasą kontrolera (Scene) osadzającą instancje trzech innych klas (Customer, Clerk, Parrot) i kierującą nimi. Klasy osadzonych instancji mogą również brać udział w hierarchii dziedziczenia. Kompozycja i dziedziczenie są często tak samo użytecznymi sposobami nadawania kodowi struktury umożliwiającej jego ponowne wykorzystanie

Należy utworzyć kod klas Pythona implementujący tę strukturę za pomocą kompozycji. Kod obiektu Scene ma definiować metodę `action` i osadzać instancje klas `Customer` (klient), `Clerk` (sprzedawca) oraz `Parrot` (papuga) — wszystkie trzy powinny definiować metodę `line` wyświetlającą unikalny komunikat. Osadzone obiekty mogą dziedziczyć po wspólnej klasie nadzędnej definiującej `line` i udostępniać komunikat `text` lub samodzielnie definiować `line`. W rezultacie klasy powinny działać w następujący sposób.

```

% python
>>> import parrot

>>> parrot.Scene().action()                  # Aktywacja
zagnieżdżonych obiektów
customer: "To już ekspapuga!"
clerk: "Nie, wcale nie..."
parrot: None

```

Programowanie zorientowane obiektowo według mistrzów

Kiedy prowadzę kursy z Pythona, nieodmiennie spotykam się z tym, że w połowie kursu osoby, które używały już programowania zorientowanego obiektowo w przeszłości, z

łatwością nadążają za materiałem, natomiast pozostałe osoby zaczynają się nudzić (lub nawet przysypiają). Cel tej technologii nie jest dla nich oczywisty.

W książce takiej, jak ta, mam luksus dołączenia materiału takiego, jak przegląd szerokiej perspektywy z rozdziału 26. czy stopniowe wprowadzenie z rozdziału 28. Szczerze zachęcam każdego, kto zaczyna czuć, że programowanie zorientowane obiektywnie to jakieś komputerowe trele-morele, by raz jeszcze powrócił do tego fragmentu książki. Choć programowanie to dodaje znacznie więcej struktury, niż dodają poznane wcześniej generatory, opiera się na pewnej magii (przeszukiwanie dziedziczenia i pierwszy argument specjalny), której początkujący programiści mogą nie rozumieć.

Na moich kursach — by zachęcić osoby poczynające (a czasem uniemożliwić im drzemkę) — znany jestem z tego, że zatrzymuję się i pytam ekspertów spośród publiczności, po co używają programowania zorientowanego obiektywnie. Odpowiadzi, jakich mi udzielają, mogą pomóc objaśnić nieco cel programowania zorientowanego obiektywnie osobom, dla których zagadnienie to jest nowością.

Poniżej, jedynie z drobnymi upiększeniami, znajdują się powody wykorzystywania programowania zorientowanego obiektywnie podawane przez moich studentów przez te wszystkie lata.

Ponowne wykorzystanie kodu

Ten powód jest prosty (i jest też najważniejszą przyczyną korzystania z programowania zorientowanego obiektywnie). Dzięki obsłudze dziedziczenia klasy pozwalają nam programować za pomocą dostosowywania kodu do własnych potrzeb zamiast rozpoczynania każdego projektu od podstaw.

Hermetyzacja

Opakowywanie szczegółów implementacji za interfejsami obiektów izoluje użytkowników klasy od zmian kodu.

Struktura

Klasy udostępniają nowe zakresy lokalne, co pozwala zminimalizować konflikty między nazwami zmiennych. Są również naturalnym miejscem, w którym pisze się i wyszukuje kod implementacyjny, a także gdzie zarządza się stanem obiektów.

Utrzymywanie

Klasy w sposób naturalny promują faktoryzację kodu, co pozwala nam na minimalizowanie jego powtarzalności. Dzięki strukturze oraz obsłudze ponownego wykorzystania zazwyczaj zmienić musimy jedynie jedną kopię kodu.

Spójność

Klasy oraz dziedziczenie pozwalają na implementowanie wspólnych interfejsów i tym samym na taki sam wygląd oraz działanie kodu. Ułatwia to debugowanie, zrozumienie kodu oraz jego utrzymywanie.

Polimorfizm

To raczej właściwość programowania zorientowanego obiektywnie, a nie powód korzystania z niego, ale dzięki obsłudze uogólnienia kodu polimorfizm sprawia, że kod ten staje się bardziej elastyczny i nadający się do różnych zastosowań, dzięki czemu można go wykorzystywać wielokrotnie.

Pozostałe

I na koniec, oczywiście, jeden z powodów wykorzystywania programowania zorientowanego obiektywnie podany przez pewnego studenta, który brzmi: ponieważ wygląda to dobrze w naszym CV! No dobrze, powód ten dodałem jako dowcip, jednak jeśli planujemy w dzisiejszych czasach pracować w przemyśle informatycznym, znajomość programowania zorientowanego obiektywnie jest istotna.

Należy również pamiętać o tym, co napisałem na początku tej części książki — programowanie zorientowane obiektowo docenia się wtedy, gdy się go trochę poużywa. Należy wybrać jakiś projekt, przestudiować większe przykłady, samodzielnie wykonać ćwiczenia — zrobić cokolwiek, co zwiększy nasze obycie z kodem zorientowanym obiektowo. Na pewno nam się to opłaci.

[1] Dla przykładu: w liczczącej 1600 stron książce *Programming Python* poświęconej programowaniu aplikacji i będącej kontynuacją niniejszej książki wykorzystywana jest wyłącznie wersja 3.x. Nigdzie nie są w niej wykorzystywane opisane w tym rozdziale narzędzia w nowym stylu ani nie było potrzeby dostosowania jej do nich, a mimo to książka pokazuje, jak tworzyć rozbudowane interfejsy graficzne, strony WWW, systemy, bazy danych i aplikacje przetwarzające tekst. Prezentowane są w niej głównie proste kody, wykorzystujące wbudowane typy i biblioteki, bez niejasnych i ezoterycznych rozszerzeń programowania obiektowego. Stosowane klasy są relatywnie proste, ilustrują strukturę oraz faktoring kodu i prawdopodobnie lepiej odzwierciedlają praktyczne programowanie niż klasy opisane w tym podręczniku. Oznacza to, że wiele zaawansowanych narzędzi do programowania obiektowego w Pythonie jest dość sztucznych, bardziej związanych ze strukturą języka niż jego praktycznym przeznaczeniem. W tej książce pozwalam sobie na luksus zredukowania zestawu narzędzi do zakresu niezbędnego do tworzenia użytecznego kodu. Nic nie stoi na przeszkodzie, aby tak robić, o ile jakiś programista nie wpadnie na pomysł użycia tajemnej funkcjonalności języka!

[2] Począwszy od tego rozdziału, interaktywne listingi nie zawierają pustych wierszy, a szesnastkowe adresy obiektów są skrócone do 32 bitów. Celem jest zmniejszenie ich wielkości i poprawa czytelności listingów. Zakładam, że do tego miejsca te drobne szczegóły nie miały znaczenia.

[3] Oba artykuły są po części prywatnymi opiniami, ale warto je przeczytać. Tytuł pierwszego został zmieniony na „Funkcja super jest fajna, ale lepiej jej nie używać” (<https://fuhm.net/super-harmful>). Natomiast drugi, pomimo swego subiektywnego tonu, w jakiś sposób trafił do oficjalnego podręcznika Pythona. Odnośnik do tego artykułu znajduje się w podręczniku. Lepiej byłoby, gdyby różne opinie były bardziej równomierne reprezentowane w dokumentacji narzędzi lub całkowicie pominięte. Podręczniki do Pythona to nie jest miejsce dla prywatnych opinii i subiektywnej propagandy!

[4] Cytat pochodzi z filmu *Monty Python i Święty Graal*. Polskie tłumaczenie w wersji Tomasza Beksińskiego, udostępnione w serwisie *Monty Python's Modrzew* (<http://www.modrzew.stopklatka.pl/>) — przyp. tłum.

Część VII Wyjątki oraz narzędzia

Rozdział 33. Podstawy wyjątków

Niniejsza część książki poświęcona jest *wyjątkom* (ang. *exception*), czyli zdarzeniom, które mogą modyfikować przebieg sterowania w programie. W Pythonie wyjątki wywoływane są automatycznie w momencie wystąpienia błędów i mogą być wywoływanie oraz przechwytywane przez nasz kod. Są przetwarzane przez cztery instrukcje omówione w tej części książki, z których pierwsza ma dwa warianty (omówione tutaj osobno), a ostatnia była rozszerzeniem opcjonalnym aż do Pythona 2.6 oraz 3.0.

try/except

Przechwytuje wyjątki wywołane przez Pythona lub przez nas i pozwala sobie z nimi poradzić.

try/finally

Wykonuje działania oczyszczające bez względu na to, czy wyjątki wystąpią, czy też nie.

raise

Ręczne wywołanie wyjątku w kodzie.

assert

Warunkowe wywołanie wyjątku w kodzie.

with/as

Implementuje menedżery kontekstu w Pythonie 2.6 oraz 3.0 (w wersji 2.5 opcjonalne).

Temat ten został odłożony aż do prawie samego końca książki, ponieważ do tworzenia kodu własnych wyjątków musimy znać klasy. Z kilkoma wyjątkami (celowa gra słów) — zobaczymy, że obsługa wyjątków w Pythonie jest prosta, ponieważ w dużej mierze jest ona zintegrowana z samym językiem jako kolejne narzędzie wyższego poziomu.

Po co używa się wyjątków

Mówiąc w skrócie, wyjątki pozwalają nam wyskoczyć z dowolnie dużych części programu. Rozważmy hipotetycznego robota wytwarzającego pizzę, o którym mówiliśmy wcześniej w książce. Założymy, że pomysł ten bierzemy całkiem na poważnie i rzeczywiście zbudujemy taką maszynę. By wytworzyć pizzę, nasz kulinarny automat musi wykonać plan, który musielibyśmy zaimplementować jako program napisany w Pythonie. Musiałby on przyjąć zamówienie, przygotować ciasto, dodać składniki nadzienia, upiec ciasto i tak dalej.

Założymy teraz, że coś pójdzie nie tak na etapie „pieczenie ciasta”. Być może piekarnik jest uszkodzony, być może robot źle obliczył swój zasięg i spontanicznie się zapalił. Chcemy oczywiście móc szybko przeskoczyć do kodu obsługującego taki stan. Ponieważ nie mamy już nadziei na skończenie pizzy w tak niezwykłych okolicznościach, równie dobrze możemy porzucić cały plan.

Właśnie na to pozwalają nam wyjątki — możemy w jednym kroku przeskoczyć do programu obsługi wyjątku, porzucając wszystkie wywołania funkcji rozpoczęte od momentu wejścia do

programu obsługi wyjątku. Kod z programu obsługi wyjątku może następnie odpowiedzieć w odpowiedni sposób na zgłoszony wyjątek (na przykład wzywając straż pożarną!).

Wyjątek można sobie wyobrazić jako typ ustrukturyzowanej instrukcji „super-goto”. *Program obsługi wyjątków* (instrukcja `try`) pozostawia znacznik i wykonuje jakiś kod. Gdzieś dalej w programie zgłaszany jest wyjątek sprawiający, że Python wraca do znacznika, porzucając wszystkie aktywne funkcje, które były wywołane po opuszczeniu znacznika. Protokół ten udostępnia spójny sposób reagowania na niezwykłe zdarzenia. Co więcej, ponieważ Python natychmiast przeskakuje do instrukcji programu obsługi, kod może być prostszy — zazwyczaj nie ma konieczności sprawdzania kodów statusu po każdym wywołaniu funkcji, która potencjalnie może zawieść.

Role wyjątków

W programach napisanych w Pythonie wyjątki są zazwyczaj wykorzystywane do różnych celów. Poniżej znajduje się spis ich najważniejszych ról.

Obsługa błędów

Python zgłasza wyjątki za każdym razem, kiedy w czasie wykonywania programu znajduje w nim błąd. Możemy przechwytywać i odpowiadać na błędy w kodzie lub ignorować zgłoszane wyjątki. Jeśli błąd jest ignorowany, do gry wkracza domyślna obsługa wyjątków Pythona — zatrzymuje ona program i wyświetla komunikat o błędzie. Jeśli takie zachowanie nam nie odpowiada, musimy zapisać w kodzie instrukcję `try`, która przechwyci wyjątek i pozwoli go obsłużyć. Po wykryciu błędu Python przeskoczy do programu obsługi `try`, a program wznowi wykonywanie po `try`.

Powiadomienia o zdarzeniach

Wyjątki można również wykorzystać do sygnalizowania poprawnych warunków bez konieczności przekazywania flag wyników w programie lub jawnego ich sprawdzania. Procedura wyszukiwania może na przykład zgłosić wyjątek dla niepowodzenia, zamiast zwracać liczbowy kod wyniku (i mieć nadzieję, że kod nigdy nie będzie miał poprawnego wyniku).

Obsługa przypadków specjalnych

Czasami jakiś warunek może występować tak rzadko, że trudno jest uzasadnić przekształcanie kodu w taki sposób, by go obsługiwał. Często możemy wyeliminować kod specjalnych przypadków, obsługując je zamiast tego w programach obsługi wyjątków na wyższych poziomach. W podobny sposób używa się na etapie tworzenia kodu instrukcji `assert` w celu sprawdzenia, czy zostały spełnione oczekiwane warunki.

Działania końcowe

Jak zobaczymy, instrukcja `try/finally` pozwala nam zagwarantować, że wymagane operacje czasu zakończenia zostaną wykonane bez względu na obecność lub nieobecność wyjątków w programach. Alternatywnym rozwiązaniem jest użycie nowej instrukcji `with`, o ile dany obiekt ją obsługuje.

Nietypowy przebieg sterowania

I wreszcie, ponieważ wyjątki są rodzajem operacji „`goto`” wysokiego poziomu, możemy ich użyć jako podstawy do implementacji egzotycznego przebiegu programu. Choć na przykład Python oficjalnie nie obsługuje nawracania (ang. *backtracking*), można je w tym języku programowania zaimplementować za pomocą wyjątków, a także niewielkiej ilości logiki obsługującej służącej do rozwinięcia przypisania^[1]. W Pythonie nie ma instrukcji „`goto`” (na całe szczęście!), jednak wyjątki mogą czasami pełnić te same role, na przykład można ich używać do wychodzenia z zagnieżdżonych pętli.

Kilka powyższych ról zostało już opisanych wcześniej, a typowe przypadki użycia wyjątków będą przedstawione w dalszej części książki. Teraz zaczniemy od przyjrzenia się kilku narzędziom do obsługi wyjątków.

Wyjątki w skrócie

W porównaniu z innymi kluczowymi możliwościami języka, z jakimi spotkaliśmy się w książce, wyjątki są w Pythonie dość mało kłopotliwym narzędziem. Ponieważ są tak proste w użyciu, przejdźmy od razu do przykładu.

Domyślny program obsługi wyjątków

Załóżmy, że piszemy kod następującej funkcji.

```
>>> def fetcher(obj, index):
...     return obj[index]
...
```

W funkcji tej nie ma nic szczególnego — indeksuje ona obiekt dla przekazanej wartości indeksu. W normalnej operacji zwraca ona wynik tej całkowicie poprawnej składni.

```
>>> x = 'mielonka'
>>> fetcher(x, 3)                                # Jak x[3]
'l'
```

Jeśli jednak użyjemy w funkcji indeksu większego niż długość łańcucha znaków, to w czasie próby wykonania `obj[index]` zgłoszony zostanie wyjątek. Python wykrywa indeksowanie poza granicami sekwencji i raportuje je, zgłaszając (wywołując) wbudowany wyjątek `IndexError`.

```
>>> fetcher(x, 8)                                # Domyślny program obsługi
wyjątków – interfejs powłoki
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in fetcher
      IndexError: string index out of range
```

Ponieważ nasz kod nie przechwytuje tego wyjątku w sposób jawnego, przechodzi z powrotem do najwyższego poziomu programu i wywołuje *domyślny program obsługi wyjątków*, który po prostu wyświetla standardowy komunikat o błędzie. W tym miejscu książki widzieliśmy już kilka standardowych komunikatów o błędach. Obejmują one zgłoszony wyjątek wraz ze *śladem stosu* (ang. *stack trace*) — listą wszystkich wierszy oraz funkcji aktywnych w momencie, kiedy nastąpił wyjątek.

Powyższy tekst komunikatu o błędzie został wyświetlony przez Pythona w wersji 3.3. W różnych wydaniach Pythona, a także typach powłoki interaktywnej może się on nieznacznie różnić, dlatego nie przywiązuje nadmiernej wagi do jego formy ani w tej książce, ani w swoim kodzie. Kiedy kod wpisujemy w sesji interaktywnej w prostym interfejsie powłoki, plik to po prostu `<stdin>`, co oznacza standardowy strumień wejścia.

W przypadku pracy w powłoce interaktywnej GUI IDLE nazwą pliku jest <pyshell> i wyświetlane są także wiersze kodu źródłowego. W żadnym z przypadków numerowanie wierszy nie jest zbyt znaczące, jeśli nie mamy pliku (ciekawsze komunikaty o błędach zobaczymy nieco później w tej części książki).

```
>>> fetcher(x, 8)                                # Domyślny program obsługi wyjątków –
graficzny interfejs IDLE
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    fetcher(x, 8)
  File "<pyshell#3>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

W bardziej realistycznym programie uruchomionym poza sesją interaktywną po wyświetleniu komunikatu o błędzie domyślny program obsługi na najwyższym poziomie programu natychmiast kończy również program. Takie działanie ma sens dla prostych skryptów — błędy często są krytyczne i najlepsze, co możemy zrobić, kiedy wystąpią, to sprawdzenie standardowego komunikatu o błędzie.

Przechwytywanie wyjątków

Czasami jednak nie o to nam chodzi. Programy serwera zazwyczaj muszą na przykład pozostać aktywne nawet po błędach wewnętrznych. Jeśli nie chcemy stosować domyślnego zachowania wyjątków, powinniśmy opakować wywołanie w instrukcję `try` w celu samodzielnego ich przechwytywania.

```
>>> try:
...     fetcher(x, 8)
... except IndexError:                         # Przechwycenie wyjątku i wznowienie
    działania
...     print('mam wyjątek')
...
mam wyjątek
>>>
```

Teraz Python przeskakuje do naszego *programu obsługi* (bloku znajdującego się pod częścią `except`, który nazywa zgłoszony wyjątek) automatycznie, kiedy wyjątek jest wywoływany w czasie wykonywania bloku instrukcji `try`. Kiedy pracujemy w sesji interaktywnej, jak powyżej, po wykonaniu części `except` trafiemy z powrotem do zachęty Pythona. W bardziej realistycznym programie instrukcje `try` nie tylko przechwytyują błędy, ale również pomagają sobie z nimi *poradzić*.

```
>>> def catcher():
...     try:
...         fetcher(x, 8)
...     except IndexError:
```

```

...     print('mam wyjątek')
...     print('kontynuuuję')

...
>>> catcher()
mam wyjątek
kontynuuuję
>>>

```

Tym razem jednak po przechwyceniu i obsłużeniu wyjątku program kontynuuje wykonywanie po całej instrukcji `try`, która przechwyciła błąd — stąd otrzymujemy komunikat „kontynuuuję”. Nie widzimy standardowego komunikatu o błędzie, a program normalnie kontynuuje swoje działanie.

Zwróć uwagę, że w Pythonie nie ma możliwości *powrotu* do kodu, który zgłosił wyjątek (choć oczywiście można ponownie uruchomić kod, aby dojść do tego samego punktu). Po przechwyceniu wyjątku wykonywany jest kod znajdujący się poza blokiem `try`, a nie poniżej instrukcji, która ten wyjątek zgłosiła. W Pythonie czyszczona jest cała pamięć wykorzystywana przez funkcje, z których nastąpiło wyjście w wyniku wystąpienia wyjątku (w tym przykładzie `fetcher`), i nie można wznowić ich wykonywania. Instrukcja `try` zarówno przechwytuje wyjątki, jak również określa, od którego miejsca wznowiane jest wykonywanie programu.



W opisywanych przykładach wielokrotnie pojawia się prefiks „...” w instrukcji `try` użytej na najwyższym poziomie. Kod skopiowany do schowka i wklejony do konsoli nie będzie działał, chyba że zostanie użyty w zagnieżdżonej funkcji lub klasie. Instrukcja `except` i inne instrukcje muszą być odpowiednio wyrównane z instrukcją `try`, dlatego nie są przed nimi umieszczane wiodące spacje niezbędne do zilustrowania struktury wcięć. Aby uruchomić kod, musisz po prostu wpisać go ręcznie lub kopować pojedynczo instrukcje znajdujące się wierszach poprzedzonych znakami „...”.

Zgłaszanie wyjątków

Dotychczas pozwalaliśmy Pythonowi zgłaszać wyjątki za nas, popełniając błędy (tym razem celowo!), jednak nasze skrypty mogą również same zgłaszać wyjątki. Oznacza to, że wyjątki mogą być zgłaszane przez Pythona lub nasz program; mogą być przechwytywane lub nie. By ręcznie wywołać wyjątek, wystarczy wykonać instrukcję `raise`. Wyjątki wywoływane przez użytkownika są przechwytywane w taki sam sposób jak wywoywane przez Pythona. Poniższy kod może nie być najbardziej użytecznym programem Pythona w historii, ale dobrze ilustruje, o co nam chodzi.

```

>>> try:
...     raise IndexError                               # Ręczne wywołanie wyjątku
... except IndexError:
...     print('mam wyjątek')
...
mam wyjątek

```

Jak zwykle, jeśli wyjątek zgłoszony przez użytkownika nie zostanie przechwycony, jest on przekazywany do domyślnego programu obsługi wyjątków na najwyższym poziomie programu,

a program kończy się ze standardowym komunikatem o błędzie.

```
>>> raise IndexError  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError
```

Jak zobaczymy w kolejnym rozdziale, do wywołania wyjątku można także wykorzystać instrukcję `assert` — jest ona warunkowym odpowiednikiem `raise`, wykorzystywany głównie na cele debugowania kodu w trakcie programowania:

```
>>> assert False, 'Nikt nie spodziewa się hiszpańskiej inkwizycji!'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AssertionError: Nikt nie spodziewa się hiszpańskiej inkwizycji!
```

Wyjątki zdefiniowane przez użytkownika

Instrukcja `raise` przedstawiona w poprzednim podrozdziale powoduje zgłoszenie wbudowanego wyjątku Pythona. Jak dowiemy się w dalszej części książki, możemy również definiować nowe własne wyjątki, specyficzne dla naszego programu. Wyjątki zdefiniowane przez użytkowników można tworzyć za pomocą *klas* dziedziczących po wbudowanej klasie wyjątków — zazwyczaj jest to klasa o nazwie `Exception`.

```
>>> class Bad(Exception):          # Wyjątek zdefiniowany przez  
    użytkownika  
    ...     pass  
    ...  
>>> def doomed():  
    ...     raise Bad()           # Zgłoszenie instancji  
    ...  
>>> try:  
    ...     doomed()  
    ... except Bad:              # Przechwycenie nazwy klasy  
    ...     print('przechwycenie Bad')  
    ...  
przechwycenie Bad  
>>>
```

Jak się przekonasz w dalszej części rozdziału, za pomocą instrukcji `as` i `except` można uzyskać dostęp do obiektu wyjątku. Wyjątki oparte na klasach pozwalają skryptom na budowanie kategorii wyjątków, dziedziczenie zachowania i dodawanie metod i informacji o stanie. Można również dostosowywać treści komunikatów wyświetlanych w przypadku nieprzechwycenia wyjątku, jak niżej:

```
>>> class Career(Exception):
```

```
def __str__(self): return 'Zostałem więc kelnerem...'

>>> raise Career()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.Career: Zostałem więc kelnerem...
>>>
```

Działania końcowe

Instrukcja `try` mogą wreszcie określać, co stanie się na końcu — zawierać blok `finally`. Wygląda on tak jak blok `except`, jednak kombinacja `try/finally` określa działania końcowe, jakie zawsze zostaną wykonane przy wychodzeniu, bez względu na to, czy w bloku `try` wystąpi wyjątek.

```
>>> try:
...     fetcher(x, 3)
... finally:                                # Działania końcowe
...     print('po pobraniu')
...
'l'
po pobraniu
>>>
```

W powyższym kodzie, jeśli blok `try` zakończy się bez wystąpienia wyjątku, blok `finally` nadal zostanie wykonany, a program będzie kontynuowany po całej instrukcji `try`. W tym przypadku instrukcja ta wydaje się dość głupia — równie dobrze moglibyśmy wpisać instrukcję `print` zaraz po wywołaniu funkcji i całkowicie pominąć instrukcję `try`.

```
fetcher(x, 3)
print('po pobraniu')
```

Takie rozwiązanie jest jednak problematyczne. Jeśli wywołanie funkcji zwróci wyjątek, nigdy nie wykonamy instrukcji `print`. Połączenie `try/finally` pozwala uniknąć tej pułapki — jeśli wyjątek wystąpił w bloku `try`, bloki `finally` wykonywane są, kiedy program jest rozwijany.

```
>>> def after():
...     try:
...         fetcher(x, 8)
...     finally:
...         print('po pobraniu')
...     print('po try?')
...
>>> after()
po pobraniu
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
    File "<stdin>", line 2, in fetcher
IndexError: string index out of range
>>>
```

W kodzie tym nie otrzymujemy komunikatu „po try?”, ponieważ sterowanie normalnie nie jest wznawiane po bloku `try/finally`, kiedy wystąpi wyjątek. Zamiast tego Python przeskakuje z powrotem do działania z `finally`, a następnie *przekazuje* wyjątek w górę do poprzedniego programu obsługi (w tym przypadku będzie to domyślny program obsługi wyjątków znajdujący się na najwyższym poziomie). Jeśli zmodyfikujemy to wywołanie wewnątrz funkcji, tak by nie wywoływało ono wyjątku, kod z `finally` nadal jest wykonywany, jednak po instrukcji `try` program jest kontynuowany.

```
>>> def after():
...     try:
...         fetcher(x, 3)
...     finally:
...         print('po pobraniu')
...         print('po try?')
...
>>> after()
po pobraniu
po try?
>>>
```

W praktyce kombinacje `try/except` przydają się do przechwytywania błędów i radzenia sobie z nimi, natomiast `try/finally` są użyteczne w celu zagwarantowania, że działania końcowe zostaną wykonane bez względu na to, czy w kodzie bloku `try` wystąpi jakiś wyjątek. Możemy na przykład użyć `try/except` do przechwycenia błędów zgłaszanych przez kod importowany z biblioteki zewnętrznej, natomiast `try/finally` wykorzystać w celu zapewnienia, że wywołania zamkajające pliki lub połączenia z serwerem zawsze zostaną wykonane. Praktyczne przykłady takich sytuacji zobaczymy później w tej części książki.

Choć służą one do zupełnie innych celów, od Pythona 2.5 możemy teraz mieszać części `except` oraz `finally` w jednej instrukcji `try` — `finally` wykonywane jest przy wyjściu bez względu na to, czy wyjątek został zgłoszony, a także bez względu na to, czy wyjątek został przechwycony przez część `except`.

Znaczenie sprawdzania błędów

Jednym ze sposobów przekonania się o przydatności wyjątków jest porównanie kodu napisanego w Pythonie oraz w językach niemających wyjątków. Jeśli na przykład chcemy pisać rozbudowane programy w języku C, musimy sprawdzać zwracane wartości oraz kody statusu po każdej operacji, która potencjalnie mogłaby pójść nie tak, i przekazywać wyniki testów w miarę działania programu.

```
doStuff()
```

```

{
    if (doFirstThing() == ERROR)          # Program w języku C
        return ERROR;                    # Wykrywa błędy wszędzie
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}

```

Tak naprawdę prawdziwe programy w języku C często zawierają tyle samo kodu sprawdzającego błędy co kodu wykonującego prawdziwą pracę. Jednak w Pythonie nie musimy być aż tak metodyczni (i neurotyczni!). Zamiast tego możemy opakować dowolnie duże fragmenty programu w programy obsługi wyjątków i po prostu napisać te części, które wykonują prawdziwą pracę, zakładając, że wszystko będzie w porządku.

```

def doStuff():                      # Kod w Pythonie
    doFirstThing()                  # Nie przejmujemy się wyjątkami
    doNextThing()                  # więc nie musimy ich wykrywać
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()                  # Tu martwimy się o wyniki
    except:                        # więc jest to jedyne miejsce,
        które musimy sprawdzić
        badEnding()
    else:
        goodEnding()

```

Ponieważ sterowanie przeskakuje do programu obsługi natychmiast, kiedy tylko wystąpi wyjątek, nie ma konieczności angażowania całego kodu w stanie na straży i ostrzeganie przed błędami. Co więcej, ponieważ Python wykrywa błędy automatycznie, nasz kod zazwyczaj w ogóle nie musi ich sprawdzać. W rezultacie wyjątki pozwalają nam w dużej

mierze ignorować nietypowe przypadki, dzięki czemu nie trzeba rozpraszać się pisaniem kodu sprawdzającego błędy.

Jak dowiemy się z kolejnego rozdziału, Python 2.x oraz 3.x udostępnia alternatywę dla kombinacji `try/finally` przy użyciu pewnych typów obiektów. Instrukcja `with/as` wykonuje logikę menedżera kontekstu obiektu, gwarantując tym samym wystąpienie działania końcowego:

```
>>> with open('lumberjack.txt', 'w') as file:      # Zawsze zamyka plik przy
   ...     wyjściu
   ...     file.write('Modrzew!\n')
```

Choć opcja ta wymaga mniejszej liczby wierszy kodu, ma ona zastosowanie jednie przy przetwarzaniu pewnych typów obiektów, dlatego kombinacja `try/finally` jest bardziej uniwersalną strukturą końcową. Z drugiej strony `with/as` może także wykonywać działania początkowe i obsługuje zdefiniowany przez użytkownika kod menedżera kontekstu.

Podsumowanie rozdziału

To już większość kwestii związanych z wyjątkami — jest to naprawdę proste narzędzie.

Podsumowując, wyjątki są w Pythonie narzędziem sterowania przebiegiem wysokiego poziomu. Mogą być zgłaszane przez Pythona lub nasze własne programy. W obu przypadkach mogą być ignorowane (w celu wywołania domyślnego komunikatu o błędzie) lub przechwytywane za pomocą instrukcji `try` (by można je było przetworzyć za pomocą kodu). Instrukcja `try` ma dwa formaty logiczne, które od Pythona 2.5 można łączyć — jeden obsługuje wyjątki, a drugi wykonuje kod końcowy bez względu na to, czy wyjątek wystąpił. Instrukcje `raise` oraz `assert` Pythona wywołują wyjątki na żądanie (dotyczy to zarówno wyjątków wbudowanych, jak i nowych, zdefiniowanych przez nas za pomocą klas). Instrukcja `with/as` jest alternatywnym sposobem upewnienia się, że działania końcowe zostaną wykonane dla obsługujących je obiektów.

W pozostałej części książki uzupełnimy pewne szczegóły dotyczące przedstawionych tu instrukcji, sprawdzimy, jakie elementy mogą pojawiać się pod `try`, i omówimy obiekty wyjątków oparte na klasach. Kolejny rozdział zaczniemy od bliższego przyjrzenia się przedstawionym tutaj instrukcjom. Zanim jednak przejdziemy dalej, zalecam wykonanie quizu podsumowującego omówione tutaj podstawy.

Sprawdź swoją wiedzę — quiz

1. Należy wymienić trzy cele, jakie pełni przetwarzanie wyjątków.
2. Co stanie się z wyjątkiem, kiedy nie zrobimy nic specjalnego w celu obsłużenia go?
3. W jaki sposób skrypt może sobie poradzić po wystąpieniu wyjątku?
4. Należy podać dwa sposoby wywoływania wyjątków w skrypcie.
5. Należy podać dwa sposoby określania działań, które zostaną wykonane w momencie zakończenia skryptu, bez względu na to, czy wystąpił wyjątek.

Sprawdź swoją wiedzę – odpowiedzi

1. Przetwarzanie wyjątków przydaje się w obsłudze błędów, działań końcowych oraz powiadomieniach o zdarzeniach. Może także uprosić obsługę przypadków specjalnych i można je wykorzystać do implementacji alternatywnego przebiegu sterowania. Ogólnie przetwarzanie wyjątków pozwala zminimalizować ilość kodu sprawdzającego błędy, jakiego może wymagać program. Ponieważ wszystkie błędy odfiltrowywane są do programów obsługi wyjątków, być może nie będziemy musieli sprawdzać wyniku każdego z wykonywanych działań.
2. Wszystkie nieprzechwycone wyjątki odfiltrowywane są do domyślnego programu obsługi wyjątków, jaki Python udostępnia na najwyższym poziomie skryptu. Ten program obsługi wyjątków wyświetla znane komunikaty o błędach i zamyka skrypt.
3. Jeśli chcemy uniknąć domyślnego wyświetlania komunikatu o błędzie i zamknięcia, możemy zapisać w kodzie instrukcje `try/except`, przechwytyjące zgłoszone wyjątki i radzące sobie z nimi. Po przechwyceniu wyjątek ten zostaje zakończony, a program może być kontynuowany.
4. Instrukcje `raise` oraz `assert` można wykorzystać do wywołania wyjątku w dokładnie taki sam sposób, jak gdyby był on zgłoszony przez samego Pythona. Z reguły można także zgłosić wyjątek, popełniając błąd programistyczny, jednak nie jest to zazwyczaj cel twórcy programu.
5. Instrukcję `try/finally` można wykorzystać do upewnienia się, że pewne działania zostaną wykonane po wyjściu z bloku kodu, bez względu na to, czy kod ten zgłasza wyjątek, czy też nie. Do upewnienia się, że wykonane zostaną działania końcowe, może służyć także instrukcja `with/as`, jednak tylko przy przetwarzaniu typów obiektów, które to obsługują.

[1] Prawdziwe nawracanie nie jest częścią języka Python. Nawracanie polega na anulowaniu przed wyjściem z bloku kodu wszystkich wykonanych operacji. W Pythonie tak nie jest. Zmienne przypisane pomiędzy czasem wejścia do instrukcji `try` a czasem zgłoszenia wyjątku nie są przywracane do wcześniejszych wartości. Nawet funkcje i wyrażenia generatorów przedstawione w rozdziale 20. nie realizują prawdziwego nawracania po prostu odpowiadają one na żądania `next(G)`. Osoby zainteresowane tym zagadnieniem odsyłam do książek poświęconych sztucznej inteligencji lub językom Prolog albo Icon.

Rozdział 34. Szczegółowe informacje dotyczące wyjątków

W poprzednim rozdziale przyjrzaliśmy się nieco działaniu instrukcji powiązanych z wyjątkami. W tym zagłębimy się w to zagadnienie znacznie bardziej — w niniejszym rozdziale znaleźć można bardziej formalne wprowadzenie do składni przetwarzania wyjątków w Pythonie. Zapoznamy się tutaj zwłaszcza ze szczegółowymi informacjami dotyczącymi instrukcji `try`, `raise`, `assert` oraz `with`. Jak zobaczymy, choć instrukcje te są stosunkowo proste, oferują duże możliwości w zakresie radzenia sobie z wyjątkami w kodzie napisanym w Pythonie.



Jedna uwaga proceduralna: zagadnienie wyjątków w ostatnich latach uległo pewnym zmianom. Od Pythona 2.5 część `finally` może pojawiać się w tej samej instrukcji `try` co części `except` oraz `else` (wcześniej nie można ich było ze sobą łączyć). W Pythonie 3.0 oraz 2.6 nowa instrukcja menedżera kontekstu `with` stała się oficjalną częścią języka, natomiast wyjątki definiowane przez użytkownika muszą teraz być instancjami klas, dziedziczącymi po wbudowanej klasie nadzędnej wyjątków. W Pythonie 3.x można także napotkać lekko zmodyfikowaną składnię instrukcji `raise` i części `except`, dostępną już w wersjach 2.6 i 2.7.

W tym wydaniu książki skupię się na stanie wyjątków z Pythona 2.x i 3.x, jednak ponieważ przez jakiś czas będzie możliwa w kodzie spotkać oryginalne techniki stosowane we wcześniejszych wersjach, przy okazji wskażę również, jak bardzo zmieniły się niektóre rzeczy w tej dziedzinie.

Instrukcja `try/except/else`

Skoro zapoznaliśmy się już z podstawami, czas na szczegóły. W poniższym omówieniu najpierw postaram się przedstawić `try/except/else` oraz `try/finally` jako osobne instrukcje, ponieważ w wersjach Pythona starszych od 2.5 pełnią one inne role i nie można ich łączyć. Jak wspomniałem w uwadze wyżej, w Pythonie 2.5 i nowszych wersjach `except` oraz `finally` mogą być łączone w jednej instrukcji `try`. Konsekwencje tej zmiany wyjaśnię po osobnym omówieniu dwóch oryginalnych form.

Instrukcja `try` jest instrukcją złożoną. Rozpoczyna się ona od wiersza nagłówka `try`, po którym następuje blok (normalnie) wciętych instrukcji, a później jedna lub większa liczba części identyfikujących wyjątki, jakie mają być przechwycone, oraz opcjonalna część `else` na końcu. Słowa `try`, `except` oraz `else` są ze sobą związane za pomocą indentacji na tym samym poziomie (czyli wyrównaniu w pionie). Podsumowując, ogólny format tej instrukcji w Pythonie 3.x przedstawiony jest poniżej.

```
try:  
    <instrukcje>                                # Wykonanie najpierw tego głównego  
    działania  
  
except <nazwa1>:
```

```

<instrukcje>           # Wykonane, jeśli nazwa1 zostanie zgłoszona
w bloku try

except (nazwa2, nazwa3):
    <instrukcje>       # Wykonane, kiedy wystąpi jeden z tych
wyjątków

except <nazwa4> as <dane>:
    <instrukcje>       # Wykonane, jeśli nazwa4 zostaje zgłoszona
i zgłoszona zostaje instancja

except:
    <instrukcje>       # Wykonane dla wszystkich (pozostałych)
zgłoszonych wyjątków

else:
    <instrukcje>       # Wykonane, jeśli żaden wyjątek nie został
zgłoszony w bloku try

```

W instrukcji tej blok pod nagłówkiem `try` reprezentuje *podstawowe działanie* tej instrukcji — kod, który próbujemy wykonać i objąć obsługą błędów. Części `except` definiują *programy obsługi* dla wyjątków zgłoszanych w bloku `try`, natomiast część `else` (o ile jest obecna) udostępnia program obsługę, który wykonamy, jeśli nie wystąpią *żadne* wyjątki. Wpis `<dane>` związany jest z możliwościami instrukcji `raise` i klasami wyjątków, które omówimy w dalszej części rozdziału.

Jak działa instrukcja `try`

A oto, jak działają instrukcje `try`. Po wejściu do instrukcji `try` Python oznacza kontekst bieżącego programu, tak by mógł do niego wrócić, jeśli wystąpi wyjątek. Instrukcje zagnieździone pod nagłówkiem `try` wykonywane są jako pierwsze. To, co się dzieje później, zależy od tego, czy w czasie wykonywania instrukcji z bloku `try` zgłoszane są wyjątki i spełnione zdefiniowane warunki:

- Jeśli wyjątek *wystąpi*, kiedy wykonywane są instrukcje z bloku `try`, Python przeskakuje z powrotem do `try` i wykonuje instrukcje znajdujące się pod pierwszą częścią `except`, odpowiadającą zgłoszonemu wyjątkowi. Sterowanie podejmowane jest pod całą instrukcją `try` po wykonaniu bloku `except` (o ile blok ten nie powoduje zgłoszenia kolejnego wyjątku). Po zakończeniu tego bloku wykonywany jest kod znajdujący się poniżej całej instrukcji `try` (chyba że w bloku `except` zostanie zgłoszony kolejny wyjątek; w takim wypadku cały opisany proces jest wykonywany od nowa).
- Jeśli wyjątek *występuje* w bloku `try`, a nie pasuje do *żadnej* części `except`, jest on przesyłany do góry aż do najbliższej pasującej instrukcji `try`. Gdy poszukiwanie zakończy się na najwyższym poziomie kodu i instrukcja nie zostanie znaleziona, wtedy Python kończy działanie programu i wyświetla domyślny komunikat o błędzie.
- Jeśli żaden wyjątek *nie pojawi się* w czasie wykonywania instrukcji znajdujących się pod nagłówkiem `try`, Python wykonuje instrukcje znajdujące się pod wierszem `else` (o ile jest on obecny), a sterowanie wznowiane jest pod całą instrukcją `try`.

Innymi słowy, części `except` przechwytyują wszystkie wyjątki, jakie wystąpią w czasie wykonywania bloku `try`, natomiast część `else` wykonywana jest tylko wtedy, gdy w czasie wykonywania bloku `try` nie pojawi się żaden wyjątek. Wyjątki są porównywane z nazwami podanymi w instrukcji `except` z uwzględnieniem relacji pomiędzy superklasami opisanymi w następnym rozdziale. Pusta instrukcja `except` (bez nazwy) odpowiada każdemu wyjątkowi (lub wszystkim pozostałym).

Części `except` są *wyspecjalizowanymi* programami obsługi wyjątków — przechwytyują wyjątki pojawiające się jedynie wewnątrz instrukcji powiązanego z nimi bloku `try`. Ponieważ jednak instrukcje bloku `try` mogą wywoływać funkcje umieszczone w innych miejscach programu, źródło wyjątku może się znajdować poza samą instrukcją `try`.

Blok `try` może być dowolnie duży. Może nawet zawierać kolejne instrukcje `try`, które są wykonywane w pierwszej kolejności, gdy pojawi się wyjątek. Więcej informacji na ten temat pojawi się przy okazji omawiania *zagłędzania* instrukcji `try` w rozdziale 36.

Części instrukcji `try`

Kiedy piszemy kod instrukcji `try`, pod nagłówkiem instrukcji `try` mogą się pojawić różne części. W tabeli 34.1 przedstawiono wszystkie możliwe formy; musimy użyć przynajmniej jednej z nich. Z niektórymi spotkaliśmy się już wcześniej — jak wiemy, `except` przechwytuje wyjątki, `finally` działa przy wychodzeniu, a `else` wykonywane jest, kiedy nie wystąpiły żadne wyjątki.

Tabela 34.1. Formy części instrukcji `try`

Forma części	Interpretacja
<code>except:</code>	Przechwytuje wszystkie (lub wszystkie pozostałe) typy wyjątków
<code>except nazwa:</code>	Przechwytuje jedynie wymieniony wyjątek
<code>except nazwa as wartość:</code>	Przechwytuje wymieniony wyjątek oraz jego instancję
<code>except (nazwa1, nazwa2):</code>	Przechwytuje dowolny z wymienionych wyjątków
<code>except (nazwa1, nazwa2) as wartość:</code>	Przechwytuje dowolny z wymienionych wyjątków i jego instancję
<code>else:</code>	Wykonywane, jeśli wyjątki nie zostały zgłoszone
<code>finally:</code>	Zawsze wykonuje ten blok

Części z dodatkowym wpisem w postaci `as wartość` omówimy, kiedy spotkamy instrukcję `raise`. Umożliwiają one dostęp do obiektów zgłaszanych jako wyjątki.

Przechwytywanie wybranych i wszystkich wyjątków

Pierwszy oraz czwarty wpis z tabeli 34.1 są jednak nowe.

- `except bez nazwy wyjątku` (forma `except:`) przechwytuje *wszystkie* wyjątki niewymienione wcześniej w instrukcji `try`,
- `except z listą wyjątków w nawiasach` (forma `except (e1, e2, e3):`) przechwytuje *dowolny* z wymienionych wyjątków.

Ponieważ Python szuka dopasowania wewnątrz instrukcji `try`, przeglądając części `except od góry do dołu`, wersja w nawiasach ma taki sam skutek jak wymienienie każdego wyjątku w osobnej części `except`, jednak ciało tej instrukcji wystarczy zapisać tylko raz. Poniżej znajduje się przykład działania instrukcji z wieloma częściami `except`, który pokazuje, jak bardzo specyficzne mogą być nasze programy obsługi wyjątków.

```
try:  
    action()
```

```

except NameError:
    ...
except IndexError:
    ...
except KeyError:
    ...
except (AttributeError, TypeError, SyntaxError):
    ...
else:
    ...

```

W powyższym przykładzie, jeśli wyjątek zostanie zgłoszony w czasie wykonywania wywołania funkcji `action`, Python powraca do `try` i szuka pierwszego `except` wymieniającego zgłoszony wyjątek. Części `except` sprawdzane są od góry do dołu i od lewej strony do prawej. Python wykonuje instrukcje znajdujące się pod pierwszym `except`, jakie pasuje. Jeśli żadna część `except` nie pasuje, wyjątek przekazywany jest poza tę instrukcję `try`. Warto zauważyć, że `else` wykonywane jest tylko wtedy, gdy w funkcji `action` nie wystąpi *żaden* wyjątek. Nie jest wykonywane natomiast, kiedy zgłoszony zostaje wyjątek niepasujący do żadnego `except`.

Przechwytywanie wszystkich wyjątków: pusta instrukcja except i klasa Exception

Jeśli naprawdę potrzebujemy uniwersalnej części `except` przechwytyjącej wszystko, przyda nam się puste `except`.

```

try:
    action()
except NameError:
    ...                                     # Obsługa NameError
except IndexError:
    ...                                     # Obsługa IndexError
except:
    ...                                     # Obsługa wszystkich pozostałych
    wyjątków
else:
    ...                                     # Obsługa przypadku bez wyjątku

```

Pusta część `except` przechwytuje wszystko i pozwala na to, by nasze programy obsługi wyjątków były tak ogólne lub tak specyficzne, jak tylko sobie życzymy. W niektórych sytuacjach ta forma może być wygodniejsza od podania wszystkich możliwych wyjątków w instrukcji `try`. Przykładowo poniższy kod przechwytuje wszystkie wyjątki bez wymieniania jakichkolwiek.

```

try:
    action()
except:

```

```
...  
wyjątki
```

Przechwytuje wszystkie możliwe

Puste `except` wiąże się z pewnymi problemami projektowymi. Choć jest wygodne, może przechwytywać nieoczekiwane wyjątki systemowe niezwiązane z naszym kodem, a także może niechcący przechwytywać wyjątki przeznaczone dla innych programów obsługi. Przykładowo systemowy sygnał zakończenia programu oraz naciśnięcie klawiszy `Ctrl+C` powoduje zgłoszenie wyjątku i zazwyczaj chcemy, by się one powiodły. Struktura ta może jednak również przechwytywać prawdziwe błędy programistyczne, dla których najprawdopodobniej chcemy zobaczyć komunikaty o błędach. Powróćmy do tej pułapki pod koniec tej części książki. Na razie powiem tylko tyle: należy z tej opcji korzystać ostrożnie.

W Pythonie 3.x wprowadzono alternatywne rozwiązanie jednego z tych problemów. Przechwycenie wyjątku o nazwie `Exception` daje prawie taki sam efekt jak puste `except`, jednak ignoruje wyjątki powiązane z systemowymi wyjściemi z programu.

```
try:  
    action()  
except Exception:  
    ...
```

Przechwytuje wszystkie możliwe wyjątki z

wyjątkiem wyjść z programu

Tajniki obsługi wyjątku w powyższej formie będą opisane w następnej części rozdziału, poświęconej klasom wyjątków. W skrócie można powiedzieć, że polega to na sprawdzaniu, czy klasa wyjątku jest zgodna z podklassą wskazaną w instrukcji `except`. Klasa `Exception` jest nadklassą dla wszystkich wyjątków, które można przechwytywać w opisany wyżej sposób. Jest to tak samo wygodne rozwiązanie jak pusta instrukcja `except`, jednak unika się ryzyka przechwycenia wyjątków systemowych. Mimo wszystko istnieje niebezpieczeństwo przeoczenia błędów programistycznych.



Uwaga dotycząca wersji: więcej informacji o instrukcji `as` znajdziesz w dalszej części rozdziału, poświęconej instrukcji `raise`. Począwszy od wersji Pythona 3.x, wymagane jest użycie składni `except Wyjątek as Wartość`: (wymienionej w tabeli 34.1 i wykorzystywanej w niniejszej książce) w miejsce starszej formy `except Wyjątek, Wartość`:. Ta druga postać nadal jest dostępna w Pythonie 2.6 i 2.7 (jednak nie jest zalecana) — jeśli jednak zostanie użyta, zastosowana zostanie konwersja na pierwszą postać.

Zmiana ta została dokonana w celu wyeliminowania błędów związanych z mylением tej starszej formy z zapisem z dwoma alternatywnymi wyjątkami, w Pythonie 2.6 zapisywanym jako `except (Wyjątek1, Wyjątek2)`:. Ponieważ w Pythonie 3.x obsługiwana jest jedynie postać z `as`, przecinki w części `except` zawsze oznaczają krotkę, bez względu na fakt użycia nawiasów lub ich brak, a wartości interpretowane są jako alternatywne wyjątki, które mają być przechwycone.

Jak się za chwilę przekonamy, składnia ta nie wpływa na zasadę zakresu widoczności zmiennych obowiązującą w wersji 2.x, gdzie nawet w przypadku użycia instrukcji `as` zmienna `Wartość` jest widoczna poza blokiem `except`. W wersji 3.x zmienna ta nie jest widoczna, ponieważ na końcu bloku jest usuwana.

Część `try/else`

Na pierwszy rzut oka cel części `else` nie zawsze jest oczywisty dla osób początkujących. Bez niej jednak nie da się powiedzieć (bez ustawiania i sprawdzania flag Boolean), czy przebieg

sterowania przeszedł przez instrukcję `try`, ponieważ wyjątek nie został zgłoszony lub wystąpił i został obsłużony. Niezależnie od przypadku wykonywany jest kod poza blokiem `try`:

```
try:  
    ...wykonanie kodu...  
except IndexError:  
    ...obsłuzenie wyjątku...  
# Czy trafiliśmy tutaj dlatego, że instrukcja try się nie powiodła, czy wręcz  
przeciwnie?
```

Podobnie jak `else` w pętlach sprawia, że powód wyjścia z pętli jest bardziej oczywisty, część `else` w instrukcjach `try` udostępnia składnię sprawiającą, że to, co się wydarzyło, jest jasne i jednoznaczne.

```
try:  
    ...wykonanie kodu...  
except IndexError:  
    ...obsłuzenie wyjątku...  
else:  
    ...wyjątek nie wystąpił...
```

Możemy prawie emulować część `else`, przenosząc jej kod do bloku `try`.

```
try:  
    ...wykonanie kodu...  
    ...wyjątek nie wystąpił...  
except IndexError:  
    ...obsłuzenie wyjątku...
```

Może to jednak prowadzić do nieprawidłowych klasyfikacji wyjątków. Jeśli akcja „wyjątek nie wystąpił” wywołuje błąd `IndexError`, zostanie zarejestrowana jako niepowodzenie bloku `try` i tym samym błędnie wywoła program obsługi wyjątku znajdujący się pod `try` (subtelne, ale prawdziwe!). Wykorzystując zamiast tego jawnie zdefiniowaną część `else`, sprawiamy, że logika jest bardziej oczywista, i gwarantujemy, że program obsługi `except` zostanie wykonany jedynie dla prawdziwych błędów w kodzie opakowanym za pomocą `try`, a nie dla błędów w działaniu przypadku `else`.

Przykład — zachowanie domyślne

Ponieważ przebieg sterowania w programie jest łatwiejszy do ujęcia w Pythonie niż w języku polskim, wykonajmy kilka przykładów, które będą ilustrować podstawy wyjątków.

Wspomniałem już, że wyjątki nieprzechwycone przez instrukcje `try` przechodzą w górę do najwyższego poziomu procesu Pythona i wykonują logikę domyślnej obsługi wyjątków tego języka (co oznacza, że Python kończy działający program i wyświetla standardowy komunikat o błędzie). Przyjrzyjmy się teraz przykładowi. Wykonanie poniższego pliku modułu `bad.py` generuje wyjątek dzielenia przez zero.

```
def gobad(x, y):
```

```
    return x / y

def gosouth(x):
    print(gobad(x, 0))

gosouth(1)
```

Ponieważ program ignoruje wywoływanego wyjątku, Python kończy jego działanie i wyświetla komunikat.

```
% python bad.py
Traceback (most recent call last):
  File "bad.py", line 7, in <module>
    gosouth(1)
  File "bad.py", line 5, in gosouth
    print(gobad(x, 0))
  File "bad.py", line 2, in gobad
    return x / y
ZeroDivisionError: int division or modulo by zero
```

Powyższy kod wykonałem w oknie powłoki w Pythonie 3.x. Komunikat składa się ze śladu stosu („Traceback”) oraz nazwy (i dodatkowych danych) wyjątku, który został zgłoszony. Ślad stosu wymienia wszystkie wiersze, które były aktywne w czasie, gdy wystąpił wyjątek, od najstarszych do najnowszych. Warto zauważyc, że ponieważ nie pracujemy w sesji interaktywnej, w tym przypadku informacje dotyczące pliku i numeru wiersza mogą być bardziej przydatne. Na powyższym listingu widzimy na przykład, że niepoprawne dzielenie ma miejsce w ostatnim wpisie śladu — drugim wierszu pliku *bad.py*, w instrukcji `return`^[1].

Ponieważ Python wykrywa i zgłasza wszystkie błędy w czasie wykonywania za pomocą zgłaszania wyjątków, wyjątki są nieodłącznie związane z zagadnieniami obsługi błędów oraz, szerzej, debugowania. Każdy, kto zapoznawał się z przykładami zaprezentowanymi w książce, musiał po drodze widzieć co najmniej kilka wyjątków — nawet błędy literowe generują `SyntaxError` czy inny wyjątek, kiedy plik jest importowany lub wykonywany (czyli kiedy uruchamiany jest kompilator). Domyślnie otrzymujemy przydatny sposób wyświetlania błędów, taki, jak pokazano powyżej, co pozwala nam prześledzić problem.

Często ten standardowy komunikat o błędzie jest wszystkim, co jest potrzebne do rozwiązywania problemów w kodzie. W przypadku zaawansowanych zadań związanych z debugowaniem możemy przechwytywać wyjątki za pomocą instrukcji `try` lub wykorzystać jedno z narzędzi do debugowania omówionych w rozdziale 3., do których powrócimy w rozdziale 36. (na przykład moduł `pdb` z biblioteki standardowej).

Przykład — przechwytywanie wbudowanych wyjątków

Domyślna obsługa wyjątków w Pythonie jest często dokładnie tym, czego potrzebujemy — w szczególności w przypadku kodu na najwyższym poziomie pliku skryptu błąd zazwyczaj powinien od razu zakończyć działanie programu. W wielu programach nie ma potrzeby być bardziej specyficznych w zakresie błędów w kodzie.

Czasami jednak chcemy przechwytywać błędy i radzić sobie z nimi. Jeśli nie chcemy, by nasz programkończył swoje działanie, kiedy Python zgłasza wyjątek, wystarczy przechwycić ten wyjątek, opakowując logikę programu w instrukcję `try`. Jest to ważna możliwość w programach

takich, jak serwery sieciowe, które muszą działać bez przerwy. Poniższy kod (zawarty w pliku *kaboom.py*) przechwytuje błąd `TypeError`, który Python zgłasza natychmiast, kiedy próbujemy dokonać konkatenacji listy oraz łańcucha znaków (operator `+` oczekuje tego samego typu sekwencji po obu stronach), i radzi sobie z nim.

```
def kaboom(x, y):
    print(x + y)                                # Wywołanie TypeError

try:
    kaboom([0,1,2], "mielonka")

except TypeError:                               # Przechwycenie błędu i
    poradzenie sobie z nim

    print('Witaj, świecie!')

print('wznowienie tutaj')                      # Kontynuacja bez względu na
                                                # wystąpienie wyjątku
```

Kiedy w funkcji `kaboom` wystąpi wyjątek, sterowanie przeskakuje do części `except` instrukcji `try`, która wyświetla komunikat. Ponieważ wyjątek jest „martwy” po przechwyceniu w ten sposób, program kontynuuje wykonywanie pod instrukcją `try`, a nie zostaje zakończony przez Pythona. W rezultacie kod przetwarzany jest i czyści błąd, a skrypt radzi sobie z nim:

```
% python kaboom.py
```

```
Witaj, świecie!
```

```
wznowienie tutaj
```

Warto zauważyć, że po przechwyceniu wyjątku sterowanie wznawiane jest w miejscu, w którym go przechwyciliśmy (czyli po `try`). Nie ma prostego sposobu na wrócenie do miejsca, w którym wyjątek wystąpił (w powyższym przykładzie — funkcja `kaboom`). W pewnym sensie sprawia to, że wyjątki bardziej przypominają proste przeskoki niż wywołania funkcji — nie istnieje możliwość powrócenia do kodu, który wywołał wyjątek.

Instrukcja `try/finally`

Inną odmianą instrukcji `try` jest jej specjalizacja związana z działaniami końcowymi. Jeśli w instrukcji `try` znajduje się część `finally`, Python zawsze wykonuje jej blok instrukcji przy wychodzeniu z instrukcji `try`, bez względu na to, czy w czasie wykonywania bloku `try` wystąpił wyjątek. Jej ogólna forma jest następująca.

```
try:
    <instrukcje>                                # Wykonanie najpierw tego
    działania

finally:
    <instrukcje>                                # Zawsze wykonuje ten kod na
    wyjściu
```

W tym wariantie Python rozpoczyna od wykonania bloku instrukcji powiązanego z wierszem nagłówka `try`. To, co dzieje się następnie, uzależnione jest od tego, czy w czasie bloku `try` wystąpi wyjątek.

- Jeśli w czasie wykonywania bloku `try` *nie wystąpi* wyjątek, Python przeskakuje z powrotem w celu wykonania bloku `finally`, a następnie kontynuuje wykonywanie pod instrukcją `try`.
- Jeśli w czasie wykonywania bloku `try` *wystąpi* wyjątek, Python nadal powraca i wykonuje blok `finally`, ale następnie przekazuje wyjątek wyżej — albo do znajdującej się wyżej instrukcji `try`, albo do domyślnego programu obsługi na najwyższym poziomie programu. Program nie wznawia wykonywania poniżej instrukcji `try`. Blok `finally` jest zatem wykonywany, nawet jeśli zgłoszony zostanie wyjątek, jednak w przeciwieństwie do `except`, `finally` nie kończy wyjątku — jego zgłoszenie jest kontynuowane po wykonaniu bloku `finally`.

Forma `try/finally` jest przydatna, kiedy chcemy być całkowicie pewni, że po wykonaniu jakiegoś kodu na pewno wystąpi określone działanie, bez względu na zachowanie wyjątku w programie. W praktyce pozwala to na określenie działań oczyszczających, które muszą wystąpić zawsze — takich, jak zamykanie plików czy połączenia z serwerem.

Warto zauważyć, że część `finally` nie może być wykorzystana w tej samej instrukcji `try` co `except` oraz `else` w Pythonie 2.4 oraz wcześniejszych wersjach, dlatego połączenie `try/finally` najlepiej jest sobie wyobrazić jako osobną formę instrukcji, jeśli korzystamy z tych starszych wydań Pythona. W Pythonie 2.5 i nowszych wersjach `finally` może się pojawić w tej samej instrukcji co `except` oraz `else`, dlatego dzisiaj istnieje tak naprawdę jedna instrukcja `try` z wieloma opcjonalnymi częściami (więcej informacji na ten temat niedługo). Bez względu na wykorzystywaną wersję część `finally` nadal pełni tę samą rolę — pozwala na określanie działań „czyszczących”, które muszą być wykonywane zawsze, bez względu na wystąpienie wyjątków.



Jak zobaczymy również później w niniejszym rozdziale, w Pythonie 2.6 i 3.0 instrukcja `with` oraz menedżery kontekstu udostępniają oparty na obiektach sposób wykonania podobnego działania dla sytuacji wyjścia. W przeciwieństwie do `finally` ta nowa instrukcja obsługuje również działania wykonywane na wejściu, jednak jest ograniczona w swoim zakresie do obiektów implementujących protokół menedżera kontekstu.

Przykład — działania kończące kod z użyciem `try/finally`

Z kilkoma prostymi przykładami zastosowania instrukcji `try/finally` spotkaliśmy się już w poprzednim rozdziale. Poniżej znajduje się bardziej realistyczny przykład ilustrujący typowe zastosowanie tej instrukcji.

```
class MyError(Exception): pass

def stuff(file):
    raise MyError()

file = open('data', 'w')                      # Otwarcie pliku wyjścia (tu też
może wystąpić problem)

try:
    stuff(file)                                # Zgłoszenie wyjątku
finally:
    file.close()                               # Zawsze zamykanie pliku w celu
wyczyszczenia bufora wyjścia
```

```
print('nie doszliśmy tutaj')          # Kontynuacja tutaj tylko jeśli nie
ma wyjątku
```

Kiedy funkcja z kodu powyżej zgłasza wyjątek, sterowanie przeskakuje do tyłu i wykonuje blok `finally` w celu zamknięcia pliku. Wyjątek jest następnie przekazywany albo do innej instrukcji `try`, albo do domyślnego programu obsługi na najwyższym poziomie pliku, wyświetlającego standardowy komunikat błędu i zamykającego program. Instrukcje po `try` nigdy nie zostaną wykonane. Gdyby funkcja ta *nie* zgłosiła wyjątku, program nadal wykonałby blok `finally` w celu zamknięcia pliku, jednak byłby kontynuowany poniżej całej instrukcji `try`.

W powyższym kodzie opakowaliśmy wywołanie funkcji przetwarzającej plik w instrukcję `try` z częścią `finally` celu zapewnienia, że plik zawsze będzie zamykany, a tym samym finalizowany, bez względu na ewentualne wystąpienie wyjątku. W ten sposób późniejszy kod może mieć pewność, że zawartość bufora wyjścia została zrzucona z pamięci na dysk. Podobna struktura kodu może na przykład zagwarantować, że połączenia z serwerem zawsze będą zamykane.

Jak wiemy z rozdziału 9., w standardowej wersji języka Python (CPython) obiekty plików są automatycznie zamykane w momencie czyszczenia pamięci — jest to szczególnie przydatne w przypadku plików tymczasowych, których nie przypisujemy do zmiennych. Nie zawsze jednak łatwo jest przewidzieć, kiedy nastąpi czyszczenie pamięci, zwłaszcza w większych programach lub alternatywnych implementacjach języka (jak np. Jython, PyPy), w których stosowane są inne zasady porządkowania pamięci. Instrukcja `try` sprawia, że zamykanie pliku odbywa się w sposób bardziej jawnego oraz przewidywalnego i odnosi się do określonego bloku kodu. Zapewnia zamknięcie pliku w momencie wyjścia z bloku bez względu na fakt wystąpienia wyjątku bądź jego brak.

Funkcja z tego akurat przykładu nie jest szczególnie przydatna (po prostu zgłasza ona wyjątek), jednak opakowanie wywołania w instrukcję `try/finally` jest dobrym sposobem na upewnienie się, że nasze działania kończące (czasu wyjścia) zawsze zostaną wykonane. Python zawsze wykonuje kod z bloków `finally`, bez względu na to, czy w bloku `try` pojawi się wyjątek^[2].

Warto również zauważyc, że wyjątek zdefiniowany przez użytkownika ponownie definiowany jest za pomocą klasy — jak zobaczymy w kolejnym rozdziale, wszystkie dzisiejsze wyjątki muszą być instancjami klas, zarówno w Pythonie 2.6, jak i 3.0.

Połączona instrukcja try/except/finally

We wszystkich wersjach Pythona przed wydaniem 2.5 (czyli przez mniej więcej pierwsze 15 lat jego istnienia) instrukcja `try` występowała w dwóch odmianach i tak naprawdę były to dwie odrębne instrukcje. Mogliśmy albo wykorzystać `finally` w celu zapewnienia, że zawsze wykonany zostanie kod czyszczący, albo napisać bloki `except` w celu przechwycenia określonych wyjątków i poradzenia sobie z nimi, a także opcjonalnego podania części `else`, która zostanie wykonana, jeśli nie wystąpią żadne wyjątki.

Część `finally` nie mogła zatem być łączona z `except` oraz `else`. Po części była to zasługa kwestii implementacyjnych, a po części było tak dlatego, że znaczenie połączonych instrukcji wydawało się niejasne — przechwytywanie wyjątków i radzenie sobie z nimi wydawało się koncepcją rozłączną z wykonywaniem działań czyszczących.

W Pythonie 2.5 i nowszych wersjach te dwie instrukcje zostały połączone. Dzisiaj możemy łączyć części `finally`, `except` oraz `else` w jednej instrukcji `try` — mniej więcej tak jak w języku Java. Oznacza to, że możemy już pisać instrukcje o następującej formie.

```
try:                                # Forma połączona
    podstawowe_działanie
```

```

except Wyjątek1:           # Przechwycenie wyjątku
    program_obsługi_1
except Wyjątek2:
    program_obsługi_2
...
else:                      # Kod wykonywany, gdy nie ma wyjątku
    blok_else
finally:                   # Blok finally, który jest zawsze wykonywany
    blok_finally

```

Kod z bloku *podstawowe_działanie* tej instrukcji wykonywany jest jak zwykle jako pierwszy. Jeśli kod ten zwraca wyjątek, wszystkie bloki *except* są po kolei sprawdzane, jeden po drugim, w poszukiwaniu dopasowania do zgłoszonego wyjątku. Jeśli zgłoszony wyjątek to *Wyjątek1*, wykonywany jest blok *program_obsługi_1*. Jeśli jest to *Wyjątek2*, wykonywany jest blok *program_obsługi_2* i tak dalej. Jeśli nie zostanie wykonany żaden wyjątek, wykonywany jest *blok_else*.

Bez względu na to, co stało się poprzednio, po zakończeniu głównego bloku działania i obsłużeniu zgłoszonych wyjątków wykonywany jest *blok_finally*. Kod w *blok_finally* zostanie wykonany nawet wtedy, gdy w programie obsługa wyjątku lub bloku części *else* pojawi się błąd i zgłoszony zostanie nowy wyjątek.

Jak zawsze część *finally* nie kończy wyjątku — jeśli wyjątek jest aktywny, kiedy wykonywany jest *blok_finally*, kontynuuje on bycie przekazywanym po wykonaniu kodu z *finally*, a sterowanie przeskakuje do innego miejsca w programie (innej instrukcji *try* lub domyślnego programu obsługi błędów z najwyższego poziomu). Jeśli w czasie wykonywania *finally* żaden wyjątek nie jest aktywny, sterowanie wznowiane jest po całej instrukcji *try*.

W rezultacie część *finally* wykonywana jest zawsze, bez względu na to, czy:

- wyjątek wystąpił w podstawowym działaniu i został obsłużony,
- wyjątek wystąpił w podstawowym działaniu i nie został obsłużony,
- w podstawowym działaniu nie wystąpiły żadne wyjątki,
- nowy wyjątek został wywołany w jednym z programów obsługi wyjątków.

Warto powtórzyć, że *finally* służy do określania działań czyszczących, które zawsze muszą się pojawić przy wyjściu z instrukcji *try* bez względu na to, jakie wyjątki zostały zgłoszone lub obsłużone.

Składnia połączonej instrukcji *try*

Przy takim połączeniu instrukcja *try* musi zawierać albo *except*, albo *finally*, a części te muszą mieć odpowiednią kolejność:

try -> *except* -> *else* -> *finally*

gdzie *else* i *finally* są opcjonalne i może występować zero lub większa liczba *except*; natomiast jeśli dostępne jest *else*, w kodzie musi się znaleźć przynajmniej jedno *except*. Instrukcja *try* składa się tak naprawdę z dwóch części — *except* z opcjonalnym *else* oraz *finally*.

Składnię połączonej instrukcji *try* najlepiej będzie opisać w następujący sposób (nawiasy kwadratowe oznaczają opcjonalność, a znak * oznacza „zero lub większą liczbę”).

```

try:                                # Format 1
    instrukcje
except [typ [as wartość]]:          # W Pythonie 2.x: [typ [, wartość]]
    instrukcje
[except [typ [as wartość]]:
    instrukcje]*
[else:
    instrukcje]
[finally:
    instrukcje]
try:                                # Format 2
    instrukcje
finally:
    instrukcje

```

Z uwagi na powyższe reguły część `else` może się pojawić jedynie wtedy, gdy występuje przynajmniej jedno `except`, i zawsze można mieszać ze sobą `except` oraz `finally`, bez względu na to, czy w instrukcji występuje `else`. Można także mieszać ze sobą `finally` oraz `else`, jednak tylko w sytuacji, w której w instrukcji pojawia się także `except` (choć `except` może pomijać nazwę wyjątku w celu przechwycenia wszystkiego i wykonania opisanej dalej instrukcji `raise` w celu ponownego zgłoszenia bieżącego wyjątku). Jeśli złamiemy którąś z reguł dotyczących kolejności, Python zwróci wyjątek błędu składni przed wykonaniem kodu.

Łączenie finally oraz except za pomocą zagnieżdżania

Przed wersją 2.5 możliwe było łączenie części `finally` oraz `except` w instrukcji `try` za pomocą składniowego zagnieżdżania instrukcji `try/except` w bloku `try` instrukcji `try/finally` (technikę tę omówimy dokładniej w rozdziale 36., ale opisane tu podstawy pomogą zrozumieć połączoną instrukcję `try`). Tak naprawdę poniższy kod ma taki sam efekt jak nowa, połączona forma pokazana na początku tego podrozdziału.

```

try:                                # Zagnieżdżony odpowiednik
formy połączonej
try:
    podstawowe_działanie
except Wyjątek1:
    program_obsługi_1
except Wyjątek2:
    program_obsługi_2
...
else:
    bez_błędu

```

```
finally:  
    czyszczenie
```

Blok `finally` wykonywany jest zawsze przy wychodzeniu, bez względu na to, co stało się w podstawowym działaniu i bez względu na programy obsługi błędów wykonywane w zagnieżdżonej instrukcji `try` (można prześledzić wymienione wcześniej cztery przypadki w celu sprawdzenia, czy działa to tak samo). Ponieważ `else` zawsze wymaga istnienia `except`, postać zagnieżdżona ma te same ograniczenia w zakresie łączenia co forma połączona omówiona wyżej.

Zagnieżdżony odpowiednik jest jednak bardziej niejasny i wymaga więcej kodu niż nowa, połączona forma (co najmniej o jeden czteroznakowy wiersz i dodatkowe wcięcie). Włączenie `finally` do jednej instrukcji sprawia, że jest ona łatwiejsza do pisania oraz czytania, dlatego jest dziś preferowaną techniką.

Przykład połączonego try

Poniżej znajduje się demonstracja działania połączonej formy `try` w praktyce. Poniższy plik o nazwie `mergedexc.py` zawiera cztery często spotykane sytuacje wraz z instrukcjami `print` opisującymi znaczenie każdej z nich.

```
# Plik mergedexc.py (Python 3.x i 2.x)  
  
sep = ' - ' * 32 + '\n'  
  
print(sep + 'WYJĄTEK ZGŁOSZONY I PRZECHWYCONY')  
  
try:  
    x = 'spam'[99]  
  
except IndexError:  
    print('wykonano except')  
  
finally:  
    print('wykonano finally')  
  
print('po wykonaniu')  
  
print(sep + 'WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY')  
  
try:  
    x = 'spam'[3]  
  
except IndexError:  
    print('wykonano except')  
  
finally:  
    print('wykonano finally')  
  
print('po wykonaniu')  
  
print(sep + 'WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY, WYKONANO ELSE')  
  
try:  
    x = 'spam'[3]
```

```

except IndexError:
    print('wykonano except')
else:
    print('wykonano else')
finally:
    print('wykonano finally')
print('po wykonaniu')
print(sep + 'WYJĄTEK ZGŁOSZONY, ALE NIEPRZECHWYCONY')
try:
    x = 1 / 0
except IndexError:
    print('wykonano except')
finally:
    print('wykonano finally')
print('po wykonaniu')

```

Po wykonaniu tego kodu w Pythonie 3.3 zwracane są następujące dane wyjściowe. W wersji 2.x działanie i dane wyjściowe są takie same, ponieważ wywołania `print` za każdym razem wyświetlają jeden element, jedynie treść komunikatu może się nieznacznie różnić. Należy prześledzić ten kod w celu zobaczenia, w jaki sposób obsługa wyjątków zwraca dane wyjściowe każdego z czterech testów.

c:\code> **py -3 mergedexc.py**

WYJĄTEK ZGŁOSZONY I PRZECHWYCONY

wykonano except
wykonano finally
po wykonaniu

WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY

wykonano finally
po wykonaniu

WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY, WYKONANO ELSE

wykonano else
wykonano finally
po wykonaniu

```
WYJĄTEK ZGŁOSZONY, ALE NIEPRZECHWYCONY
wykonano finally
Traceback (most recent call last):
  File "mergedexec.py", line 39, in <module>
    x = 1 / 0
ZeroDivisionError: int division or modulo by zero
```

Przykład ten wykorzystuje wbudowane operacje w podstawowym działaniu — do wywołania wyjątków lub niezrobienia tego. Opiera się na fakcie, iż Python zawsze sprawdza błędy w czasie wykonywania kodu. W kolejnym podrozdziale pokażemy, w jaki sposób można zamiast tego ręcznie wywoływać wyjątki.

Instrukcja raise

By w jawnny sposób wywołać wyjątki, wykorzystuje się instrukcję `raise`. Ich ogólna forma jest prosta — instrukcja `raise` składa się ze słowa `raise`, po którym opcjonalnie następuje klasa zgłoszanego wyjątku lub jej instancja.

```
raise <instancja>                                # Zgłoszenie instancji klasy
raise <klasa>                                    # Utworzenie i zgłoszenie
instancji klasy
raise                                                # Ponowne zgłoszenie ostatniego
wyjątku
```

Jak wspomniano wcześniej, wyjątki w Pythonie 2.6, 3.0 i nowszych wersjach są zawsze instancjami klas. Tym samym pierwsza forma instrukcji `raise` jest także najbardziej popularna — przekazujemy bezpośrednio *instancję* utworzoną albo przed instrukcją `raise`, albo wewnątrz samej tej instrukcji. Jeśli zamiast tego przekażemy *klasę*, Python wywoła klasę bez argumentów konstruktora w celu utworzenia instancji do zgłoszenia. Forma ta jest odpowiednikiem dodania nawiasów po referencji do klasy. Ostatnia forma powoduje ponowne zgłoszenie ostatnio zgłoszonego wyjątku. Jest często wykorzystywana w programach obsługi wyjątków w celu przekazania przezchwyconych wyjątków.



Uwaga dotycząca wersji: w wersji Pythona 3.x nie jest dostępna składnia `raise Wyjątek, Argumenty`, typowa dla wersji 2.x. Zamiast niej należy stosować opisaną w tej książce instrukcję `raise Wyjątek(Argumenty)` powodującą utworzenie instancji klasy. Składnia z przecinkami, właściwa dla wersji 2.x, była zgodna z niestosowanym już dziś modelem wyjątków opartych na ciągach znaków. Użyta w wersji 3.x jest przekształcana do nowej formy.
Tak jak we wcześniejszych wersjach języka, również w zapisie `raise Wyjątek` można podać nazwę klasy. W obu wersjach jest ona przekształcana do postaci `raise Wyjątek()` powodującej wywołanie konstruktora bez argumentów. Choć składnia z przecinkami nie jest już stosowana, w wersji 2.x można po instrukcji `raise` umieścić ciąg znaków lub nazwę klasy. Pierwsza możliwość została usunięta w wersji 2.6, a w wersji 2.5 nie jest zalecana. Dlatego nie została tu opisana, a jedynie krótka wspomniana w następnym rozdziale. Dzisiaj należy stosować nową składnię z nazwą klasy wyjątku.

Zgłaszanie wyjątków

By nieco lepiej wyjaśnić temat, przyjrzyjmy się kilku przykładom. W przypadku wbudowanych wyjątków dwie poniższe formy są równoważne — obie powodują zgłoszenie instancji podanej klasy wyjątku, jednak pierwsza z nich tworzy instancję w sposób niejawnny:

```
raise IndexError                                # Klasa (utworzenie instancji)  
raise IndexError()                            # Instancja (utworzona w  
                                                instrukcji)
```

Możemy także utworzyć instancję z wyprzedzeniem. Ponieważ instrukcja `raise` przyjmuje dowolny rodzaj referencji do obiektu, dwa poniższe przykłady powodują zgłoszenie wyjątku `IndexError` tak samo jak dwa poprzednie.

```
exc = IndexError()                            # Utworzenie instancji z  
wyprzedzeniem  
  
raise exc  
  
excs = [IndexError, TypeError]  
  
raise excs[0]
```

Kiedy wyjątek zostaje zgłoszony, Python przesyła zgłoszoną instancję wraz z wyjątkiem. Jeśli instrukcja `try` zawiera część `except nazwa as X:`, do zmiennej `X` zostanie przypisana instancja przekazana w instrukcji `raise`.

```
try:  
    ...  
  
    except IndexError as X:                      # Do X przypisany zostaje  
    zgłoszony obiekt instancji  
    ...
```

Część `as` jest w programie obsługa `try` opcjonalna (jeśli zostanie pominięta, instancja nie zostanie po prostu przypisana do zmiennej), jednak dołączenie jej pozwala programowi na dostęp zarówno do danych z instancji, jak i metod z klasy wyjątku.

Model ten działa tak samo w przypadku wyjątków zdefiniowanych przez użytkownika i utworzonych za pomocą klas. Poniższy kod przekazuje na przykład argumenty konstruktora klasy wyjątku, które zostają udostępnione w programie obsługi za pośrednictwem przypisanej instancji:

```
class MyExc(Exception): pass  
...  
raise MyExc('mielonka')                      # Klasa wyjątku z argumentami  
konstruktora  
...  
try:  
    ...  
  
    except MyExc as X:                        # Atrybuty instancji dostępne w  
    programie obsługi  
    print(X.args)
```

Ponieważ zagadnienie to zaczyna się jednak pokrywać z tematem kolejnego rozdziału, odłożymy na razie szczegółowe omówienie tych kwestii.

Bez względu na nazwy wyjątków zawsze są one identyfikowane przez obiekty instancji i w danym momencie aktywny może być najwyżej jeden. Po przechwyceniu przez część `except` w dowolnym miejscu programu wyjątek przestaje istnieć (to znaczy nie będzie przekazywany do innej instrukcji `try`), o ile nie zostanie zgłoszony ponownie przez kolejną instrukcję `raise` lub błąd.

Zakresy widoczności zmiennych w instrukcjach `try` i `except`

Obiekty wyjątków będą dokładniej opisane w następnym rozdziale. Teraz, ponieważ widzieliśmy już, jak stosuje się instrukcję `as` ze zmienną, możemy wreszcie zająć się tematem widoczności zmiennych, krótko wspomnianym w rozdziale 17. W języku Python 2.x zasięg zmiennej użytej w instrukcji `except` nie jest ograniczony do samej instrukcji i rozciąga się na następujący po niej blok kodu:

```
c:\code> py -2
>>> try:
...     1 / 0
... except Exception as X: # W wersji 2.x zmienna X nie jest lokalna
...     print X
...
integer division or modulo by zero
>>> X
ZeroDivisionError('integer division or modulo by zero',)
```

Tak się dzieje w wersji 2.x, gdy używa się instrukcji `as` (jak w wersji 3.x) lub starszej składni z przecinkiem:

```
>>> try:
...     1 / 0
... except Exception, X:
...     print X
...
integer division or modulo by zero
>>> X
ZeroDivisionError('integer division or modulo by zero',)
```

W wersji 3.x jest inaczej. Zmienna zawierająca odwołanie do wyjątku jest zmienną lokalną w bloku `except`. Poza tym blokiem nie jest dostępna, podobnie jak zmienna tymczasowa nie jest widoczna poza wyrażeniem pętli w wersji 3.x (w której — jak wspomniałem wcześniej — nie można też stosować składni z przecinkiem):

```
c:\code> py -3
>>> try:
```

```

...     1 / 0
... except Exception, X:
SyntaxError: invalid syntax

>>> try:
...     1 / 0
... except Exception as X: # W wersji 3.x zmienna użyta po instrukcji as jest
dostępna tylko w bloku except
...     print(X)
...
division by zero
>>> X
NameError: name 'X' is not defined

```

W odróżnieniu od zmiennych stosowanych w wyrażeniach pętlowych w wersji 3.x zmienna użyta z instrukcją `except` znika po zakończeniu wykonywania bloku. Musi tak być, ponieważ w przeciwnym wypadku zachowywane byłoby odwołanie do stosu wywołań, a co za tym idzie, zajmowane przez tę zmienną miejsce nie byłoby zwalniane podczas oczyszczania pamięci. Zmienna jest usuwana nawet wtedy, gdy używa się jej w innym miejscu. Jak zatem widać, jest to bardziej restrykcyjna zasada zarządzania zmiennymi niż w przypadku wyrażeń. Ilustruje to poniższy kod:

```

>>> X = 99
>>> try:
...     1 / 0
... except Exception as X:           # W wersji 3.x jest to zmienna lokalna,
usuwana po wyjściu z bloku!
...     print(X)
...
division by zero
>>> X
NameError: name 'X' is not defined
>>> X = 99
>>> {X for X in 'spam'}           # W wersjach 2.x i 3.x zmienna jest
lokalna, ale nie jest usuwana
{'s', 'a', 'p', 'm'}
>>> X
99

```

Z opisanych wyżej powodów należy w instrukcjach `except` stosować unikatowe nazwy zmiennych, mimo że są one lokalne. Jeżeli potrzebne jest odwołanie do instancji klasy wyjątku poza instrukcją `try`, należy po prostu tę zmienną przypisać innej zmiennej, która nie jest automatycznie usuwana:

```

>>> try:
...     1 / 0
... except Exception as X:          # Odwołanie do tej zmiennej jest usuwane.
...     print(X)
...     Saveit = X                  # Przypisanie obiektu wyjątku w celu
zachowania go na późniejszy użytk
...
division by zero
>>> X
NameError: name 'X' is not defined
>>> Saveit
ZeroDivisionError('division by zero',)

```

Przekazywanie wyjątków za pomocą raise

Instrukcja `raise` oferuje więcej możliwości, niż wiedzieliśmy do tej pory. Na przykład używa bez nazwy wyjątku ani dodatkowych danych po prostu ponownie zgłasza bieżący wyjątek. Forma ta jest zazwyczaj wykorzystywana, kiedy musimy przechwycić oraz obsłużyć wyjątek, ale nie chcemy, by wyjątek ten zginął w kodzie.

```

>>> try:
...     raise IndexError('mielonka')           # Wyjątki pamiętają
argumenty
... except IndexError:
...     print('przekazywanie')
...     raise                                # Ponowne zgłoszenie
ostatniego wyjątku
...
przekazywanie
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: mielonka

```

Wykonanie `raise` w taki sposób ponownie zgłasza wyjątek i przekazuje go do wyższego programu obsługi (lub domyślnego programu obsługi znajdującego się na najwyższym poziomie, który zatrzymuje program ze standardowym komunikatem o błędzie). Warto zwrócić uwagę na to, jak argument przekazany do klasy wyjątku pokazuje się w komunikatach o błędach. O tym, dlaczego tak się dzieje, dowiemy się w kolejnym rozdziale.

Łańcuchy wyjątków w Pythonie 3.x — `raise from`

Wyjątki czasami zgłasza się w reakcji na wystąpienie innych wyjątków czy to specjalnie, czy w wyniku pojawięcia się nowych błędów w programie. W Pythonie 3.x (ale nie w wersji 2.x), aby

mówć obsługiwać takie przypadki, można w instrukcji `raise` używać opcjonalnej części `from`:

```
raise wyjątek from inny_wyjątek
```

Kiedy zastosowana zostanie forma z `from` i jawnie użytą instrukcją `raise`, drugi wyjątek określa inną klasę lubinstancję wyjątku, dołączane do atrybutu `__cause__` zgłoszanego wyjątku. Jeśli zgłoszony wyjątek nie zostanie przechwycony, Python wyświetla oba wyjątki jako części standardowego komunikatu o błędzie:

```
>>> try:  
...     1 / 0  
... except Exception as E:  
...     raise TypeError('Źle!') from E          # Jawnie zgłoszony kolejny  
wyjątek  
...  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
ZeroDivisionError: int division or modulo by zero  
The above exception was the direct cause of the following exception:  
Traceback (most recent call last):  
  File "<stdin>", line 4, in <module>  
TypeError: Źle!
```

Kiedy wyjątek zostaje zgłoszony wewnętrz programu obsługi wyjątków, automatycznie jest wykonywana podobna procedura. Pierwszy wyjątek zostaje dołączony do atrybutu `__context__` nowego wyjątku i jest ponownie wyświetlony w standardowym komunikacie o błędzie, jeśli wyjątek ten nie zostanie przechwycony:

```
>>> try:  
...     1 / 0  
... except:  
...     badname           # Jawne zgłoszenie kolejnego wyjątku  
...  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
ZeroDivisionError: division by zero  
During handling of the above exception, another exception occurred:  
Traceback (most recent call last):  
  File "<stdin>", line 4, in <module>  
NameError: name 'badname' is not defined
```

Ponieważ dołączany obiekt wyjątku sam może zawierać połączone obiekty innych wyjątków, wynikowy łańcuch może być bardzo długi. Wyświetlany komunikat zawiera wtedy informacje o

wszystkich błędach, nie tylko o dwóch. W efekcie zarówno w przypadku jawnego, jak i niejawnego zgłoszenia wyjątku programista może poznać je wszystkie:

```
>>> try:  
...     try:  
...         raise IndexError()  
...     except Exception as E:  
...         raise TypeError() from E  
... except Exception as E:  
...     raise SyntaxError() from E  
...  
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
    IndexError
```

Powyższy wyjątek jest bezpośrednią przyczyną zgłoszenia następującego wyjątku:

```
Traceback (most recent call last):  
  File "<stdin>", line 5, in <module>  
    TypeError
```

Ten natomiast jest bezpośrednią przyczyną następnego:

```
Traceback (most recent call last):  
  File "<stdin>", line 7, in <module>  
    SyntaxError: None
```

Poniższy kod niejawnie zgłasza trzy wyjątki i wyświetla odpowiednie komunikaty:

```
try:  
    try:  
        1 / 0  
    except:  
        badname  
except:  
    open('nie_ma_pliku')
```

Tak jak jest w przypadku ujednoliconej instrukcji `try`, wyjątki są łączone podobnie jak w innych językach (np. Java lub C#, choć nie wiadomo, z którego języka została zapożyczony ten sposób). W Pythonie składnia jest jednak wciąż dość niejasna, dlatego po więcej informacji odsyłam do dokumentacji. W wersji 3.3, zgodnie z poniższą uwagą, można zapobiegać łączeniu wyjątków.

	<p><i>Blokowanie łączenia wyjątków w wersji Python 3.3: <code>raise from None</code>.</i></p> <p>Wprowadzona w tej wersji nowa składnia umożliwia stosowanie słowa kluczowego <code>None</code> jako nazwy wyjątku w instrukcji <code>raise</code>:</p>
--	---

```
    raise newexception from None
```



W ten sposób można blokować wyświetlanie kontekstu połączonych wyjątków. Dzięki temu w aplikacjach, które przekształcają typy wyjątków zawartych w łańcuchach, komunikaty o błędach są bardziej czytelne.

Instrukcja assert

Python zawiera również instrukcję `assert` będącą w pewnym sensie przypadkiem specjalnym na cele debugowania. Jest to przede wszystkim po prostu składniowy skrót dla często wykorzystywanego wzorca z instrukcją `raise`, który można sobie wyobrazić jako *warunkową instrukcję raise*. Instrukcja w poniższej formie:

```
assert <test>, <dane> # Część <dane> jest opcjonalna
```

działa tak samo jak poniższy kod:

```
if __debug__:  
    if not <test>:  
        raise AssertionError(<dane>)
```

Innymi słowy, jeśli test okaże się fałszem, Python zgłasza wyjątek; element danych (jeśli jest podany) służy jako argument konstruktora wyjątku. Tak jak wszystkie wyjątki, zgłoszony `AssertionError` zakończy działanie programu, jeśli nie przechwycimy go za pomocą instrukcji `try`; w tym drugim przypadku element danych zostanie wyświetlony jako część komunikatu o błędzie.

Dodatkowo instrukcje `assert` można usunąć z kodu bajtowego skompilowanego programu, jeśli w wierszu poleceń Pythona użyjemy opcji `-O`, tym samym optymalizując program. `AssertionError` jest wbudowanym wyjątkiem, a opcja `__debug__` jest wbudowaną nazwą, która automatycznie ustawiana jest na 1 (True), o ile nie użyjemy opcji `-O`. W celu użycia trybu optymalizacji i wyłączenia instrukcji `assert` można zastosować kod wiersza poleceń taki jak `python -O main.py`.

Przykład – wyłapywanie ograniczeń (ale nie błędów!)

Instrukcję `assert` najczęściej wykorzystuje się do weryfikowania warunków programu w czasie jego tworzenia. Po ich wyświetleniu tekst komunikatu o błędzie automatycznie zawiera informacje o wierszu w kodzie źródłowym, a także wartość podaną w instrukcji `assert`. Rozważmy poniższy plik `asserter.py`.

```
def f(x):  
    assert x < 0, 'x musi być ujemne'  
    return x ** 2  
  
% python  
>>> import asserter  
>>> asserter.f(1)  
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
  File "asserter.py", line 2, in f
    assert x < 0, 'x musi być ujemne'
AssertionError: x musi być ujemne
```

Należy pamiętać, że instrukcja `assert` przeznaczona jest przede wszystkim do wyłapywania ograniczeń zdefiniowanych przez użytkowników, a nie do przechwytywania prawdziwych błędów programistycznych. Ponieważ Python sam wyłapuje błędy programistyczne, zazwyczaj nie istnieje konieczność tworzenia instrukcji `assert` w celu wyłapywania elementów takich, jak indeksy poza granicami sekwencji, niedopasowanie typów czy dzielenie przez zero.

```
def reciprocal(x):
    assert x != 0                                # Bezużyteczna instrukcja
    assert!
    return 1 / x                                 # Python automatycznie sprawdza
    dzielenie przez zero
```

Takie instrukcje `assert` są na ogół zbędne — ponieważ Python automatycznie zgłasza wyjątki dla błędów, możemy równie dobrze pozwolić mu wykonać tę pracę za nas. Obowiązuje zasada, że nie trzeba jawnie kodować obsługi błędów we własnym programie.

Oczywiście od większości reguł są wyjątki. Jak wspomniałem wcześniej, jeżeli funkcja wykonuje długotrwałe operacje, których nie da się cofnąć w miejscu wystąpienia wyjątku, wtedy trzeba sprawdzać możliwość wystąpienia błędu. Jednak nawet w takich przypadkach należy uważać, aby testy nie były nadmiernie rozbudowane ani restrykcyjne, ponieważ kod będzie wtedy mniej użyteczny.

Inny typowy przykład użycia instrukcji `assert` został opisany w rozdziale 29., w części poświęconej abstrakcyjnej superklasie. Instrukcja `assert` została tam wykorzystana do zabezpieczenia się przed wywołaniem niezdefiniowanej metody. Jest to rzadko stosowane, ale przydatne rozwiązanie.

Menedżery kontekstu with/as

Wersje Pythona 2.6 oraz 3.0 wprowadziły nową instrukcję związaną z wyjątkami — `with` wraz z jej opcjonalną częścią `as`. Instrukcja ta została zaprojektowana do pracy z obiekttami menedżerów kontekstu obsługującymi nowy protokół oparty na metodach. Możliwość ta dostępna jest także jako opcja w Pythonie 2.5, gdzie można ją włączyć za pomocą następującej instrukcji `import`:

```
from __future__ import with_statement
```

Instrukcja `with` jest podobna do instrukcji `using` w języku C#. Menedżery kontekstu nie obciążają systemu i bardzo się przydają do grupowania innych narzędzi do obsługi wyjątków. Jest to dodatkowy i dość zaawansowany temat, kandydat na kolejny rozdział książki.

Mówiąc w skrócie, instrukcja `with/as` ma być alternatywą dla zwykłego zastosowania `try/finally`. Podobnie jak ta instrukcja, ma ona służyć do określania działań czasu zakończenia lub działań „czyszczących”, które muszą być wykonywane zawsze, gdy wyjątek wystąpi na etapie przetwarzania.

W przeciwieństwie do `try/finally`, instrukcja `with` obsługuje bogatszy protokół oparty na obiektach, pozwalając wyznaczać działania wejścia (początkowe) oraz wyjścia (końcowe) wokół bloku kodu. Z tego powodu instrukcja ta jest mniej uniwersalna, kwalifikowana jako

redundantna pod względem roli, jaką odgrywa w przerywaniu działania programu, jak również wymagająca kodowania klas w przypadku użycia obiektów, które nie obsługują jej protokołu. Z drugiej strony jednak `with` obsługuje działania wejścia, pozwala skrócić kod i w pełni zarządzać jego kontekstem zgodnie z zasadami programowania obiektowego.

Python ulepsza za pomocą menedżerów kontekstu niektóre wbudowane narzędzia, takie jak pliki zamykające się automatycznie czy blokady wątków, które automatycznie się blokują oraz odblokowują. Programiści mogą jednak sami tworzyć kod menedżerów kontekstu za pomocą własnych klas. Przyjrzyjmy się teraz bliżej instrukcji `with` i jej protokołowi.

Podstawowe zastosowanie

Podstawowy format instrukcji `with` wygląda następująco.

`with wyrażenie [as zmienna]:`

blok_with

W powyższym kodzie `wyrażenie` ma zwracać obiekt obsługujący protokół zarządzania kontekstem (więcej informacji o tym protokole za moment). Obiekt ten może również zwracać wartość, która zostanie przypisana do elementu `zmienna`, jeśli obecna jest opcjonalna część `as`.

Warto zauważyć, że do elementu `zmienna` niekoniecznie przypisujemy `wynik` elementu `wyrażenie`. Wynikiem elementu `wyrażenie` jest obiekt obsługujący protokół kontekstu, a do elementu `zmienna` można przypisać coś innego, co zostanie wykorzystane wewnątrz instrukcji. Obiekt zwracany przez `wyrażenie` może następnie wykonać kod startowy, zanim rozpoczęty zostanie `blok_with`, a także kod końcowy po wykonaniu tego bloku, bez względu na to, czy zgłosił on jakiś wyjątek.

Niektóre obiekty wbudowane Pythona zostały rozszerzone w taki sposób, by obsługiwać protokół zarządzania kontekstem, dzięki czemu mogą być wykorzystywane w połączeniu z nową instrukcją `with`. Obiekty plików (omówione w rozdziale 9.) mają teraz na przykład menedżer kontekstu, który automatycznie zamyka plik po bloku `with`, bez względu na fakt wystąpienia jakiegoś wyjątku.

```
with open(r'C:\misc\data') as myfile:  
    for line in myfile:  
        print(line)  
        ... tutaj więcej kodu...
```

W powyższym kodzie wywołanie metody `open` zwraca prosty obiekt pliku przypisany do zmiennej `myfile`. Zmienną tą możemy wykorzystać w połączeniu z normalnymi narzędziami plików — w tym przypadku iterator pliku wczytuje go wiersz po wierszu w pętli `for`.

Obiekt ten obsługuje jednak również protokół zarządzania kontekstem wykorzystywany przez instrukcję `with`. Po wykonaniu instrukcji `with` mechanizm zarządzania kontekstem gwarantuje, że obiekt pliku, do którego odnosi się zmienna `myfile`, zostanie automatycznie zamknięty, nawet jeśli pętla `for` zgłosiła wyjątek w czasie przetwarzania pliku.

Choć obiekty plików są automatycznie zamykane w momencie czyszczenia pamięci, nie zawsze takie oczywiste jest określenie, kiedy to nastąpi, szczególnie gdy używa się różnych implementacji Pythona. Instrukcja `with` w tej roli jest alternatywą pozwalającą na zapewnienie zamknięcia pliku po wykonaniu określonego bloku kodu.

Jak widzieliśmy wcześniej, podobny efekt możemy uzyskać za pomocą bardziej uniwersalnej i jawnej instrukcji `try/finally`, jednak wymaga ona czterech wierszy kodu administracyjnego w miejsce jednego, jak poniżej:

```

myfile = open(r'C:\misc\data')

try:
    for line in myfile:
        print(line)
    ... tutaj więcej kodu...

finally:
    myfile.close()

```

Nie będziemy omawiać w książce modułów wielowątkowych Pythona (więcej informacji na ten temat można znaleźć w książkach dotyczących poziomu aplikacji, takich jak *Programming Python*), jednak definiowane przez nie blokady oraz obiekty do warunkowej synchronizacji mogą także być wykorzystywane w połączeniu z instrukcją `with`, ponieważ obsługują również protokół zarządzania kontekstem. W tym przypadku na początku i na końcu bloku kodu należy umieścić instrukcje wejściowe i wyjściowe:

```

lock = threading.Lock()

with lock:
    # kluczowy fragment kodu
    ... dostęp do współdzielonych zasobów...

```

W powyższym kodzie mechanizm zarządzania kontekstem gwarantuje, że blokada jest uzyskiwana automatycznie przed wykonaniem bloku i zdejmowana po jego zakończeniu, bez względu na wynik wyjątku.

Zgodnie z informacjami z rozdziału 5. moduł `decimal` wykorzystuje menedżery kontekstu do uproszczenia zapisywania i przywracania bieżącego kontekstu dziesiętnego określającego precyzję oraz zaokrąglenie na cele obliczeń.

```

with decimal.localcontext() as ctx:
    ctx.prec = 2
    x = decimal.Decimal('1.00') / decimal.Decimal('3.00')

```

Po wykonaniu tej instrukcji stan menedżera kontekstu bieżącego wątku jest automatycznie przywracany do stanu sprzed rozpoczęcia instrukcji. By uzyskać to samo za pomocą `try/finally`, musielibyśmy wcześniej zapisać kontekst i ręcznie go przywrócić.

Protokół zarządzania kontekstem

Choć menedżery kontekstu zostały dołączone do niektórych typów wbudowanych, możemy także pisać swoje własne. W celu zaimplementowania menedżerów kontekstów klasy wykorzystują metody specjalne mieszczące się w kategorii przeciążania operatorów. Interfejs oczekiwany od obiektów wykorzystywanych w połączeniu z instrukcją `with` jest nieco skomplikowany, a większości programistów wystarczy jedynie wiedza, jak wykorzystywać istniejące menedżery kontekstu. Na potrzeby osób budujących narzędzi, które będą musiały pisać menedżery kontekstu specyficzne dla aplikacji, przyjrzyjmy się krótko temu, jak to wygląda.

Poniżej znajduje się opis działania instrukcji `with`.

1. Wyrażenie jest analizowane, w wyniku czego otrzymujemy obiekt znany jako *menedżer kontekstu* (ang. *context manager*), który musi zawierać metody `__enter__`

oraz `__exit__`.

2. Wywołana zostaje metoda `__enter__` menedżera kontekstu. Wartość przez nią zwracana jest przypisywana do zmiennej w części `as`, jeśli jest ona obecna. W przeciwnym razie jest ona usuwana.
3. Wykonywany jest kod w zagnieżdżonym bloku `with`.
4. Jeśli blok ten zwraca wyjątek, wywołana zostaje metoda `__exit__(typ, wartość, ślad)` zawierająca szczegółowy wyjątku. Warto zauważyć, że są to te same wartości, które są zwracane przez `sys.exc_info`, opisane w dokumentacji Pythona oraz nieco później w tej części książki. Jeśli metoda ta zwraca wartość będącą fałszem, wyjątek jest zgłoszany ponownie. W przeciwnym razie wyjątek jest kończony. Wyjątek normalnie powinien być zgłoszony ponownie w celu przekazania go poza instrukcję `with`.
5. Jeśli blok nie zgłasza wyjątku, metoda `__exit__` nadal jest wywoływana, jednak do jej argumentów `typ`, `wartość` oraz `ślad` przekazywane są obiekty `None`.

Przyjrzyjmy się krótskiej demonstracji działania tego protokołu. Poniższy kod (plik `withas.py`) definiuje obiekt menedżera kontekstu, który śledzi początek i koniec bloku `with` w każdej instrukcji `with`, z jaką zostanie wykorzystany.

```
class TraceBlock:  
    def message(self, arg):  
        print('wykonywanie', arg)  
    def __enter__(self):  
        print('rozpoczęcie bloku with')  
        return self  
    def __exit__(self, exc_type, exc_value, exc_tb):  
        if exc_type is None:  
            print('normalne wyjście\n')  
        else:  
            print('zgłoszenie wyjątku!', exc_type)  
        return False # Przekazanie  
    if __name__ == '__main__':  
        with TraceBlock() as action:  
            action.message('test 1')  
            print('osiągnięty')  
        with TraceBlock() as action:  
            action.message('test 2')  
            raise TypeError  
            print('nie został osiągnięty')
```

Warto zauważyć, że metoda `__exit__` tej klasy zwraca `False` w celu przekazania wyjątku. Usunięcie instrukcji `return` dałoby tutaj ten sam efekt, ponieważ domyślna wartość `None`

zwracana przez funkcję z definicji jest `False`. Warto również zwrócić uwagę na to, iż metoda `__exit__` zwraca `self` jako obiekt do przypisania do zmiennej `as`. W innych przypadkach użycia zamiast tego może zwrócić zupełnie inny obiekt.

Po wykonaniu tego kodu menedżer kontekstu śledzi początek i koniec bloku instrukcji `with` za pomocą metod `__enter__` oraz `__exit__`. Poniżej znajduje się przykład wykonania tego skryptu w Pythonie 3.x lub 2.x (jak zwykle bywa, wyświetlany komunikat może różnić się nieco w wersjach 2.x; poniższy kod działa w wersjach 2.6, 2.7 oraz 2.5 po użyciu odpowiedniej opcji).

```
c:\code> py -3 withas.py
rozpoczęcie bloku with
wykonywanie test 1
osiągnięty
normalne wyjście
rozpoczęcie bloku with
wykonywanie test 2
zgłoszenie wyjątku! <class 'TypeError'>
Traceback (most recent call last):
  File "withas.py", line 20, in <module>
    raise TypeError
TypeError
```

Menedżery kontekstu są stosunkowo zaawansowanymi narzędziami, przeznaczonymi dla osób tworzących narzędzia, dlatego pominimy tutaj szczegóły. Pełne informacje można znaleźć w dokumentacji standardowej Pythona; można się z niej na przykład dowiedzieć, że nowy moduł standardowy `contextlib` udostępnia dodatkowe narzędzia służące do tworzenia kodu menedżerów kontekstu. Dla prostszych zastosowań instrukcja `try/finally` udostępnia wystarczającą obsługę działań czasu zakończenia.

Kilka menedżerów kontekstu w wersjach 3.1, 2.7 i nowszych

W wersji 3.1 Pythona pojawiła się rozszerzona instrukcja `with`, która ostatecznie została również wprowadzona w wersji 2.7. W powyższych i następnych wersjach można za pomocą nowej składni z przecinkami podawać kilka menedżerów kontekstu (czasami określanych mianem „zagnieżdżonych”). W poniższym przykładzie działania wyjścia obu plików wykonywane są automatycznie w momencie wyjścia z bloku instrukcji — bez względu na wyniki wyjątków.

```
with open('data') as fin, open('res', 'w') as fout:
    for line in fin:
        if 'some key' in line:
            fout.write(line)
```

Można tutaj podać dowolną liczbę menedżerów kontekstu. Większa liczba elementów działa tak samo jak zagnieżdżone instrukcje `with`. W Pythonie 3.1 (oraz późniejszych wersjach) kod:

```
with A() as a, B() as b:
```

...instrukcje...

jest równoważny z następującym, działającym w wersjach 3.0 oraz 2.6:

```
with A() as a:  
    with B() as b:  
        ...instrukcje...
```

Więcej informacji na ten temat można znaleźć w dokumentacji Pythona 3.1, ale poniżej pokazane jest krótko rozszerzenie w akcji. W tym kodzie instrukcja `with` została użyta do jednoczesnego otwarcia dwóch plików, równoległego odczytania i połączenia ich zawartości oraz automatycznego zamknięcia na koniec (przy założeniu, że wymagane jest ręczne zamknięcie plików);

```
>>> with open('script1.py') as f1, open('script2.py') as f2:  
...     for pair in zip(f1, f2):  
...         print(pair)  
  
...  
('# A first Python script\n', 'import sys\n')  
('import sys                      # Załadowanie modułu biblioteki \n',  
 'print(sys.path)\n')  
('print(sys.platform)\n', 'x = 2\n')  
('print(2 ** 32)                  # Podniesienie 2 do potęgi\n', 'print(x **  
 32)\n')
```

Powyższy schemat można wykorzystać na przykład do porównywania wiersz po wierszu dwóch plików tekstowych. W tym celu należy instrukcję `print` zastąpić prostą instrukcją `if` i dodatkowo użyć funkcji `enumerate` w celu wyświetlenia numerów wierszy:

```
with open('script1.py') as f1, open('script2.py') as f2:  
    for (linenum, (line1, line2)) in enumerate(zip(f1, f2)):  
        if line1 != line2:  
            print('%s\n%s\n' % (linenum, line1, line2))
```

Opisanej wyżej techniki nie można wykorzystać w wersji CPython, ponieważ nie jest wymagane opróżnianie buforów plików, a same pliki, jeżeli są otwarte, są automatycznie zamykane w momencie usuwania obiektów. W języku CPython pliki są zamykane automatycznie, jeżeli ich równoległe odczytywanie zakoduje się w następujący, prostszy sposób:

```
for pair in zip(open('script1.py'), open('script2.py')): # Ten sam efekt:  
    # automatyczne zamknięcie plików  
  
    print(pair)
```

Z drugiej strony w innych implementacjach języka, np. PyPy czy Jython, ze względu na inne zasady oczyszczania pamięci może być wymagane jawne zwalnianie zasobów systemowych wewnętrz pętli. Poniższe rozwiązanie jest jeszcze bardziej przydatne, ponieważ automatycznie zamyka plik wyjściowy po wykonaniu instrukcji, dzięki czemu tekst znajdujący się w buforze jest natychmiast zapisywany na dysku:

```
>>> with open('script2.py') as fin, open('upper.py', 'w') as fout:  
...     for line in fin:
```

```
...         fout.write(line.upper())
...
>>> print(open('upper.py').read())
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

W obu przypadkach można po prostu (w niektórych skryptach nawet trzeba) za pomocą osobnych instrukcji otwierać pliki i zamykać je po przetworzeniu. Nie ma sensu stosować instrukcji przechwytyujących wyjątki, jeżeli w przypadku problemu program i tak musi przerwać działanie!

```
fin = open('script2.py')
fout = open('upper.py', 'w')
for line in fin:           # Ten sam efekt co poprzednio, czyli automatyczne
    zamknięcie pliku
    fout.write(line.upper())
```

Jeżeli jednak program musi kontynuować działanie po wystąpieniu wyjątku, wtedy za pomocą instrukcji `with` można taki wyjątek niejawnie przechwycić, dzięki czemu nie trzeba stosować instrukcji `try/finally` tam, gdzie wymagane jest zamknięcie pliku. Analogiczny kod bez instrukcji `with` robi to bardziej jawnie, ale jest wyraźnie dłuższy:

```
fin = open('script2.py')
fout = open('upper.py', 'w')
try:                 # Ten sam efekt, ale pliki są jawnie zamykane w przypadku
    błędów
    for line in fin:
        fout.write(line.upper())
finally:
    fin.close()
    fout.close()
```

Z drugiej strony instrukcje `try/finally` tworzą jedno narzędzie, które można stosować we wszystkich przypadkach finalizowania kodu. Natomiast instrukcja `with` jest bardziej związanym narzędziem, ale można jej używać tylko z obiektami określonych typów. Ponadto wymaga ona, aby programista przyswoił sobie zdublowaną wiedzę. Jak zawsze, jej użycie jest kwestią indywidualnych kompromisów.

Podsumowanie rozdziału

W niniejszym rozdziale bliżej przyjrzaliśmy się przetwarzaniu wyjątków, zapoznając się z instrukcjami Pythona związanymi z tym zagadnieniem — `try` służy do przechwytywania, `raise`

do wywoływania, `assert` do warunkowego wywoływania, a `with` do opakowania bloków kodu w menedżery kontekstu, które określają działania początkowe i końcowe.

Dotychczas wyjątki wydawały się raczej łatwym do zrozumienia i zastosowania narzędziem i tak rzeczywiście jest. Jedyną nieco skomplikowaną kwestią ich dotyczącą jest sposób ich identyfikowania. W kolejnym rozdziale będziemy kontynuować nasze omówienie, opisując, w jaki sposób implementuje się własne obiekty wyjątków. Jak zobaczymy, klasy już dzisiaj pozwalają na tworzenie w kodzie nowych wyjątków specyficznych dla programu. Zanim jednak przejdziemy dalej, zalecam wykonanie quizu podsumowującego omówione tutaj podstawy.

Sprawdź swoją wiedzę — quiz

1. Do czego służy instrukcja `try`?
2. Jakie są dwie najczęściej wykorzystywane odmiany instrukcji `try`?
3. Do czego służy instrukcja `raise`?
4. Do czego służy instrukcja `assert` i jaką instrukcję przypomina?
5. Do czego służy instrukcja `with/as` i jaką instrukcję przypomina?

Sprawdź swoją wiedzę — odpowiedzi

1. Instrukcja `try` przechwytuje wyjątki i pozwala sobie z nimi radzić. Określa blok kodu, który ma być wykonany, a także jeden lub większą liczbę programów obsługujących wyjątków, które mogą być zgłoszone w czasie wykonywania bloku.
2. Dwie popularne odmiany instrukcji `try` to `try/except/else` (służąca do przechwytywania wyjątków) oraz `try/finally` (służąca do określania działań czyszczących, które muszą wystąpić bez względu na to, czy wyjątek się pojawi). Do Pythona 2.4 były to osobne instrukcje, które mogły być łączone jedynie w zagnieźdzeniu składniowym. Od wersji 2.5 bloki `except` oraz `finally` mogą być łączone w jednej instrukcji, dlatego w rezultacie te dwie formy instrukcji `try` zostały połączone. W połączonej formie `finally` jest wykonywane na wyjściu z instrukcji `try` bez względu na to, jakie wyjątki mogą zostać zgłoszone lub obsłużone. W rzeczywistości połączona forma jest odpowiednikiem konstrukcji `try/except/else` zagnieżdzonej w `try/finally`. Obie formy różnią się od siebie logicznymi rolami, jakie odgrywają w kodzie.
3. Instrukcja `raise` zgłasza (wywołuje) wyjątek. Python wewnętrznie zgłasza wbudowane wyjątki dla błędów, natomiast nasze skrypty mogą za pomocą `raise` wywoływać wyjątki wbudowane lub zdefiniowane przez użytkownika.
4. Instrukcja `assert` zgłasza wyjątek `AssertionError`, jeśli warunek jest fałszem. Działa jak warunkowa instrukcja `raise` opakowana w instrukcję `if`. Można ją zablokować za pomocą opcji `-O`.
5. Instrukcja `with/as` została zaprojektowana w celu zautomatyzowania działań początkowych oraz końcowych, które muszą wystąpić wokół bloku kodu. Przypomina w przybliżeniu instrukcję `try/finally`, ponieważ jej działania wyjściowe wykonywane są bez względu na ewentualne wystąpienie wyjątku. Pozwala ona

jednak na użycie bogatszego protokołu opartego na obiektach, określającego działania wejścia *oraz* wyjścia, jak również na skrócenie kodu. Wciąż nie jest jednak uniwersalna i można ją stosować tylko z obiektami, które obsługują jej protokół. Natomiast instrukcja `try` obejmuje znacznie więcej przypadków.

[1] Jak wspomniano w poprzednim rozdziale, tekst komunikatów o błędach oraz śladów stosu z czasem i typem powłoki nieznacznie się zmienia. Nie należy się martwić, jeśli otrzymany komunikat nie jest identyczny z moim. Kiedy na przykład wykonałem to ćwiczenie w IDLE z Pythona 3.3, tekst komunikatu o błędzie pokazywał pełną ścieżkę bezwzględną do katalogu w nazwach plików.

[2] O ile oczywiście sam Python całkowicie nie przestanie działać. Zazwyczaj dobrze mu jednak wychodzi unikanie tego — dzięki sprawdzaniu możliwych błędów w trakcie wykonywania programu. Kiedy jednak mamy do czynienia z naprawdę poważną awarią, zazwyczaj jest to zaśluga błędu w kodzie dołączonego rozszerzenia w języku C, pozostającego poza zakresem Pythona.

Rozdział 35. Obiekty wyjątków

Dotychczas luźno deliberowaliśmy o tym, czym tak naprawdę *jest wyjątek*. Jak wspomniano w poprzednim rozdziale, w Pythonie 2.6 oraz 3.0 zarówno wyjątki wbudowane, jak i zdefiniowane przez użytkownika są identyfikowane przez obiekty *instancji klas*. Wyjątki są zgłaszane i eskalowane w procesie ich obsługi, a źródłowa klasa wyjątku jest porównywana z klasami wymienionymi w instrukcji `try`.

Oznacza to co prawda, że w celu zdefiniowania nowych wyjątków w programach konieczne jest korzystanie z programowania zorientowanego obiektowo (wymagające przyswojenia wiedzy, której pełne wyłożenie zostało odłożone do tego rozdziału), jednak klasy i ten typ programowania mają kilka niezaprzeczalnych zalet.

Poniżej wymieniono kilka mocnych stron wyjątków opartych na klasach:

- **Można je organizować w kategorie.** Klasy wyjątków lepiej obsługują przyszłe zmiany dzięki udostępnianiu kategorii — dodawanie nowych wyjątków w przeszłości nie będzie wymagało modyfikacji instrukcji `try`.
- **Dołączają informacje o stanie.** Klasy wyjątków udostępniają naturalne miejsce do przechowywania informacji kontekstowych, które mogą być później wykorzystane w programach obsługi `try` — mogą zawierać zarówno dołączane informacje o stanie, jak i wywoływalne metody, dostępne za pośrednictwem instancji.
- **Obsługują dziedziczenie.** Wyjątki oparte na klasach mogą uczestniczyć w hierarchiach dziedziczenia w celu uzyskania wspólnego zachowania i dostosowywania wspólnego zachowania do własnych potrzeb. Przykładowo odziedziczone metody wyświetlania mogą udostępniać wspólny wygląd i zachowanie komunikatów o błędach.

Ze względu na te zalety wyjątki oparte na klasach dobrze obsługują ewolucję programów i większych systemów. Tak naprawdę wszystkie wbudowane wyjątki identyfikowane są przez klasy i są zorganizowane w drzewo dziedziczenia — z powodów wymienionych wyżej. To samo możemy zrobić z własnymi wyjątkami zdefiniowanymi przez użytkownika.

W Pythonie 3.x wyjątki zdefiniowane przez użytkownika dziedziczą po wbudowanych klasach nadzędnych wyjątków. Ponieważ te klasy nadzędne udostępniają przydatne wartości domyślne na cele wyświetlania oraz zachowywania stanu — jak zobaczymy — tworzenie wyjątków zdefiniowanych przez użytkownika wymaga zrozumienia roli wyjątków wbudowanych.

	<p><i>Uwaga dotycząca wersji:</i> Python 2.6 oraz 3.0 wymagają definiowania wyjątków za pomocą klas. Dodatkowo wersja 3.x wymaga, by klasy wyjątków pochodziły od wbudowanej klasy nadzędnej wyjątków <code>BaseException</code> — w sposób pośredni lub bezpośredni. Jak zobaczymy, w większości programów wyjątki dziedziczą po klasie podzędnej tej klasy o nazwie <code>Exception</code>, tak by obsługiwane były przechwytyjące wszystko programy obsługi przeznaczone dla normalnych typów wyjątków — wymienienie tej klasy w programie obsługi powoduje przechwycenie wszystkich błędów, jakie powinna przechwytywać większość skryptów. W Pythonie 2.x klasyczne niezależne klasy mogą także pełnić rolę wyjątków, jednak wymagane jest, by klasy w nowym stylu pochodziły od wbudowanych klas wyjątków, tak samo jak w wersjach 3.x.</p>
---	---

Wyjątki — powrót do przeszłości

Dawno, dawno temu (no dobrze, w czasach przed Pythonem 2.6 i 3.0) wyjątki można było definiować na dwa różne sposoby. Komplikowało to instrukcje `try`, instrukcje `raise` i w ogóle Pythona. Dzisiaj można to robić tylko w jeden sposób. To dobre rozwiązanie — pozwala na usunięcie z języka różnych pozostałości, które nagromadziły się w nim z uwagi na konieczność zachowania zgodności z wcześniejszymi wersjami. Ponieważ jednak starszy model ułatwia zrozumienie, dlaczego wyjątki są dzisiaj tym, czym są, a także z uwagi na to, że nie da się całkowicie usunąć historii czegoś, co było wykorzystywane przez miliony osób w ciągu ostatniego dwudziestolecia, nasze omówienie teraźniejszości rozpoczniemy od krótkiego przyjrzenia się przeszłości.

Wyjątki oparte na łańcuchach znaków znikają

Przed Pythonem 2.6 i 3.0 wyjątki można było definiować zarówno za pomocą instancji klas, jak i obiektów łańcuchów znaków. Wyjątki oparte na łańcuchach znaków w Pythonie 2.5 zaczęły generować ostrzeżenia, a w wersjach 2.6 oraz 3.0 całkowicie zniknęły, dlatego obecnie należy korzystać z zaprezentowanych w książce wyjątków opartych na klasach. Jeśli jednak ktoś pracuje z kodem napisanym w przeszłości, nadal może natrafić na wyjątki oparte na łańcuchach znaków. Mogą się one także pojawić w artykułach i źródłach internetowych napisanych kilka lat temu (co, licząc w pythonowych latach, można uznać za całą wieczność!).

Wyjątki oparte na łańcuchach znaków były łatwe w użyciu — wystarczył dowolny łańcuch znaków. Wyjątki te dopasowywane były po tożsamości obiektu, a nie jego wartości (czyli z użyciem `is`, a nie `==`).

```
C:\misc> C:\Python25\python
>>> myexc = "Mój łańcuch znaków wyjątku"                      # Czy kiedykolwiek
byliśmy tacy młodzi?
>>> try:
...     raise myexc
... except myexc:
...     print('przechwycony')
...
przechwycony
```

Ta postać wyjątków została usunięta, ponieważ z punktu widzenia większych programów oraz utrzymania kodu nie dorównywała ona klasom. W nowych wersjach Pythona sama próba użycia wyjątku opartego na ciągu znaków powoduje zgłoszenie wyjątku:

```
C:\code> py -3
>>> raise 'spam'
TypeError: exceptions must derive from BaseException
C:\code> py -2
>>> raise 'spam'
TypeError: exceptions must be old-style classes or derived from
BaseException, ...etc
```

Choć dzisiaj nie można już używać wyjątków opartych na łańcuchach znaków, są one naturalną podstawą omówienia modelu wyjątków opartych na klasach.

Wyjątki oparte na klasach

Łańcuchy znaków były prostym sposobem definiowania wyjątków. Jak jednak wspomniano wcześniej, klasy mają pewne zalety, które zasługują na zwrócenie na nie uwagi. Co najważniejsze, pozwalają nam identyfikować *kategorie wyjątków*, które są bardziej elastyczne do wykorzystywania oraz utrzymywania od prostych łańcuchów znaków. Co więcej, klasy w naturalny sposób pozwalają na dołączanie szczegółów dotyczących wyjątków oraz obsługują dziedziczenie. Ponieważ są lepszym rozwiązaniem, teraz są one wymagane.

Odkładając na bok szczegóły dotyczące kodu, podstawowa różnica pomiędzy wyjątkami opartymi na łańcuchach znaków oraz tymi opartymi na klasach związana jest ze sposobem dopasowywania zgłaszanych wyjątków do części `except` w instrukcjach `try`.

- Wyjątki oparte na łańcuchach znaków dopasowywane były przez prostą *tożsamość obiektu* — zgłoszony wyjątek dopasowywany był do części `except` za pomocą testu `is` Pythona.
- Wyjątki oparte na klasach dopasowywane są przez *związki z klasami nadzędnymi* — zgłoszony wyjątek dopasowywany jest do części `except`, kiedy ta część `except` wymienia klasę wyjątku lub jego dowolną klasę nadzelną.

Oznacza to, że kiedy część `except` z instrukcji `try` wymienia klasę nadzelną, przechwytuje instancje tej klasy, a także instancje wszystkich jej klas podrzędnych znajdujących się niżej w drzewie klas. W rezultacie wyjątki oparte na klasach obsługują tworzenie *hierarchii wyjątków* — klasy nadzędne stają się nazwami kategorii, a klasy podrzędne stają się specyficznymi rodzajami wyjątków wewnątrz poszczególnych kategorii. Podając ogólną klasę nadzelną wyjątków, część `except` może przechwycić całą kategorię wyjątków — dopasowana zostanie dowolna bardziej specyficzna klasa podrzędna.

W przypadku wyjątków opartych na łańcuchach znaków taka koncepcja nie istniała. Ponieważ były one dopasowywane po prostej tożsamości obiektu, nie istniał żaden bezpośredni sposób organizowania wyjątków w bardziej elastyczne kategorie czy grupy. W rezultacie programy obsługi wyjątków były łączone ze zbiorami wyjątków w sposób utrudniający wszelkie modyfikacje.

Oprócz kwestii kategorii wyjątki oparte na klasach lepiej obsługują *informacje o stanie wyjątku* (dołączane do instancji) i pozwalają wyjątkom na uczestniczenie w *hierarchiach dziedziczenia* (w celu uzyskania wspólnego zachowania). Ponieważ oferują one wszystkie zalety klas i ogólnie programowania zorientowanego obiektywnego, stanowią poważną alternatywę dla niedziałającego już modelu wyjątków opartych na łańcuchach znaków kosztem niewielkiej ilości dodatkowego kodu.

Tworzenie klas wyjątków

Przyjrzyjmy się teraz przykładowi, który pokaże nam, jak tak naprawdę w praktyce działają wyjątki. W poniższym pliku `classexc.py` definiujemy klasę nadzelną o nazwie `General` oraz dwie klasy podrzędne `Specific1` i `Specific2`. Przykład ten ilustruje pojęcie kategorii wyjątków — `General` to nazwa kategorii, a jej dwie klasy podrzędne są specyficznymi typami wyjątków wewnątrz tej kategorii. Programy obsługi przechwytyujące `General` przechwycą również wszystkie klasy podrzędne tej kategorii, w tym `Specific1` oraz `Specific2`.

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass
```

```

def raiser0():
    X = General()                                # Zgłoszenie instancji klasy
    nadrzędnej

    raise X

def raiser1():
    X = Specific1()                            # Zgłoszenie instancji klasy
    podrzędnej

    raise X

def raiser2():
    X = Specific2()                            # Zgłoszenie instancji innej
    klasy podrzędnej

    raise X

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General:                         # Dopasowanie General lub
        któreś z jej podklas

        import sys
        print('przechwycono:', sys.exc_info()[0])

```

C:\python30> **python classexc.py**

```

przechwycono: <class '__main__.General'>
przechwycono: <class '__main__.Specific1'>
przechwycono: <class '__main__.Specific2'>

```

Kod ten jest w większości zrozumiały, jednak warto zanotować kilka uwag dotyczących jego implementacji.

Klasa nadziedziona Exception

Klasy wykorzystywane do budowania drzew kategorii wyjątków mają niewiele wymagań — tak naprawdę w powyższym przykładzie są w większości puste, a ich ciała zawierają jedynie instrukcje `pass`. Warto jednak zwrócić uwagę na to, że klasa najwyższego poziomu dziedziczy tutaj po wbudowanej klasie `Exception`. Jest to wymagane w Pythonie 3.x; Python 2.x pozwala, by role wyjątków spełniały także klasyczne niezależne klasy, jednak wszystkie klasy w nowym stylu muszą pochodzić od wbudowanych klas wyjątków, tak jak w wersji 3.x. Ponieważ `Exception` udostępnia nam pewne przydatne zachowania, z którymi spotkamy się później (choć tutaj z tego nie korzystamy), w obu nowszych wersjach Pythona dobrze jest, by wyjątki dziedziczyły po tej klasie.

Zgłaszanie instancji

W powyższym kodzie wywołujemy klasy w celu utworzenia *instancji* na potrzeby instrukcji `raise`. W modelu wyjątków opartych na klasach zawsze zgłaszamy i przechwytyujemy obiekt instancji klasy. Jeśli w instrukcji `raise` wymienimy nazwę klasy bez nawiasów, Python wywoła klasę bez argumentu konstruktora w celu utworzenia instancji za nas. Instancje wyjątków można tworzyć przed wykonaniem instrukcji `raise`, tak jak w kodzie powyżej, lub wewnątrz tej instrukcji.

Przechwytywanie kategorii wyjątków

Powyższy kod zawiera funkcje zgłaszające instancje wszystkich trzech klas jako wyjątki, a także instrukcję `try` najwyższego poziomu, która wywołuje funkcje i przechwytuje wyjątki `General`. Ta sama instrukcja `try` przechwytuje również oba wyjątki szczególne, ponieważ są one klasami podrzędnymi `General`.

Szczegóły wyjątku

Program obsługuje wyjątku w kodzie powyżej wykorzystuje wywołanie `sys.exc_info` — jak zobaczymy w następnym rozdziale, w ten sposób możemy w uniwersalny sposób uzyskać dostęp do ostatnio zgłoszonego wyjątku. Mówiąc w skrócie, pierwszy element w wyniku jest klasą zgłoszonego wyjątku, natomiast drugi to zgłoszona instancja. Ogólnie w części `except` takiej jak powyższa, przechwytyująca wszystkie klasy w kategorii, `sys.exc_info` jest jednym ze sposobów ustalenia, co dokładnie się stało. W tym akurat przypadku odpowiada to pobraniu atrybutu `__class__` instancji. Jak zobaczymy w kolejnym rozdziale, rozwiązanie z `sys.exc_info` jest często wykorzystywane w połączeniu z pustą częścią `except` przechwytyującą wszystkie błędy.

Ostatnia uwaga wymaga wyjaśnienia. Kiedy wyjątek zostaje przechwycony, możemy być pewni, że zgłoszona instancja jest instancją klasy wymienionej w części `except` lub jednej z jej bardziej specyficznych klas podrzędnych. Z tego powodu atrybut `__class__` instancji podaje także typ instancji. Poniższy wariant kodu (plik `classexc2.py`) działa na przykład tak samo jak poprzedni przykład. Różni się tym, że instrukcja `except` jest rozszerzona o instrukcję `as` przypisującą zmiennej instancję klasy zgłoszanego wyjątku:

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass
def raiser0(): raise General()
def raiser1(): raise Specific1()
def raiser2(): raise Specific2()
for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General as X:                                # X to zgłoszona
        instancja
        print('przechwycono:', X.__class__)            # To samo co
        sys.exc_info()[0]
```

Ponieważ atrybut `__class__` można wykorzystać w ten sposób w celu ustalenia określonego typu zgłoszonego wyjątku, `sys.exc_info` bardziej przydaje się w przypadku pustych części `except`, które nie miałyby w inny sposób dostępu do instancji lub jej klasy. Co więcej, prawdziwe programy zazwyczaj *nie powinny musieć się martwić* o to, jaki wyjątek został zgłoszony — wywołując metody instancji w sposób ogólny, automatycznie wyzwalamy działanie przeznaczone dla zgłoszonego wyjątku.

Więcej informacji na ten temat oraz na temat `sys.exc_info` znajdzie się w kolejnym rozdziale. Osoby, które nie pamiętają już, co w instancji oznacza `__class__`, odsyłam do rozdziału 29. oraz szóstej części książki. Natomiast w poprzednim rozdziale opisana została użycia w powyższym przykładzie instrukcja `as`.

Do czego służą hierarchie wyjątków

Ponieważ w ostatnim przykładzie istnieją tylko trzy możliwe wyjątki, niezbyt dobrze oddaje on użyteczność wyjątków opartych na klasach. Tak naprawdę ten sam efekt możemy uzyskać, tworząc listę nazw wyjątków w nawiasach wewnętrznych części `except`.

```
try:  
    func()  
  
except (General, Specific1, Specific2):  
    # Przechwycenie dowolnego  
    z tych wyjątków  
  
    ...
```

Takie rozwiązanie działało także w przypadku niedziałającego już modelu wyjątków opartych na łańcuchach znaków. W przypadku dużych lub wysokich hierarchii wyjątków łatwiej może być jednak przechwytywać kategorie za pomocą klas, niż wymieniać każdy element kategorii w jednej części `except`. Co jednak ważniejsze, kategorie wyjątków możemy rozszerzać, dodając nowe klasy podrzędne bez zakłócania działania istniejącego kodu.

Załóżmy na przykład, że tworzymy w Pythonie bibliotekę numeryczną, z której będzie korzystała większa liczba osób. Kiedy piszemy bibliotekę, identyfikujemy dwie kwestie, które mogą pójść nie tak z liczbami w naszym kodzie — dzielenie przez zero oraz przepelnienie liczbowe. Dokumentujemy je jako dwa wyjątki, które może zgłosić biblioteka.

```
# Plik mathlib.py  
  
class Divzero(Exception): pass  
  
class Oflow(Exception): pass  
  
def func():  
  
    ...  
  
    raise Divzero()  
  
    ... dalsza część kodu ...
```

Kiedy teraz ktoś będzie używał biblioteki, zazwyczaj opakuje wywołania do naszych funkcji lub klas w instrukcje `try` przechwytyujące nasze dwa wyjątki (jeśli nie przechwycą one naszych wyjątków, wyjątki z biblioteki kończą działanie ich kodu).

```
# Plik client.py  
  
import mathlib  
  
try:  
    mathlib.func(...)  
  
except (mathlib.Divzero, mathlib.Oflow):  
    ...zgłoszenie i poradzenie sobie z wyjątkiem...
```

Takie rozwiązanie działa dobrze i wiele osób zaczyna korzystać z naszej biblioteki. Po sześciu miesiącach decydujemy się ją zmodyfikować (jak to często mają w zwyczaju programiści). Przy okazji identyfikujemy nowy element, który może pójść nie tak — niedomiar — i dodajemy go jako nowy wyjątek.

```
# Plik mathlib.py
```

```
class Divzero(Exception): pass  
class Oflow(Exception): pass  
class Uflow(Exception): pass
```

Niestety, kiedy publikujemy ponownie kod, tworzymy problem w utrzymywaniu dla użytkowników. Jeśli w jawnym sposobie wymienili oni nasze wyjątki, teraz muszą wrócić do kodu i zmodyfikować go w każdym miejscu, w którym wywołuje on naszą bibliotekę, tak by dodać również nazwę nowego wyjątku.

```
# Plik client.py  
  
try:  
    mathlib.func(...)  
  
except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):  
    ...zgłoszenie i poradzenie sobie z wyjątkiem...
```

Może to nie być koniec świata. Jeśli nasza biblioteka jest wykorzystywana tylko wewnętrznie, sami możemy wprowadzić te zmiany. Możemy również opublikować skrypt w Pythonie, starający się naprawić taki kod automatycznie (zapewne miałby on tylko kilka wierszy i przynajmniej czasami by się nie mylił). Gdyby jednak większa liczba osób musiała modyfikować wszystkie swoje instrukcje `try` za każdym razem, gdy zmienimy zbiór wyjątków, nie byłaby to najlepsza polityka udostępniania aktualnień.

Użytkownicy mogą próbować unikać tej pułapki, tworząc puste części `except` przechwytyujące *wszystkie* możliwe wyjątki.

```
# Plik client.py  
  
try:  
    mathlib.func(...)  
  
except:  
    # Przechwytywanie tutaj  
    wszystkiego  
  
    ...zgłoszenie i poradzenie sobie z wyjątkiem...
```

Takie obejście może jednak przechwycić więcej, niż należy — elementy takie, jak błędy braku pamięci, sekwencje przerwania wpisane z klawiatury (`Ctrl+C`), wyjście z systemu, a nawet błędy literowe w kodzie własnych instrukcji `try` użytkowników wywołują wyjątki, a wolelibyśmy, by nie były one przechwytywane i błędnie klasyfikowane jako błędy biblioteki. Użycie klasy `Exception` rozwiązuje wprawdzie ten problem, ale skutkuje też przechwytywaniem — czyli w rzeczywistości maskowaniem — błędów w kodzie.

Tak naprawdę w tej sytuacji użytkownicy chcą przechwytywać i radzić sobie jedynie z określonymi wyjątkami, które definiuje i powinna zgłaszać biblioteka. Jeśli w trakcie wywołania biblioteki pojawi się jakiś inny wyjątek, najprawdopodobniej będzie to błąd w samej bibliotece (i czas skontaktować się z jej sprzedawcą!). Z reguły w programach obsługi wyjątków lepiej jest być szczególnym niż ogólnym (do kwestii tej powróćmy w kolejnym rozdziale w podrozdziale poświęconym pułapkom związanym z wyjątkami) [1].

Co zatem należy zrobić? Hierarchie wyjątków opartych na klasach pozwalają całkowicie zażegnać ten problem. Zamiast definiować wyjątki biblioteki jako zbiór autonomicznych klas, należy ułożyć je w drzewo klas ze wspólną klasą nadzjącą obejmującą całą kategorię.

```
# Plik mathlib.py  
  
class NumErr(Exception): pass
```

```

class Divzero(NumErr): pass
class Oflow(NumErr): pass
...
def func():
    ...
    raise DivZero()
... dalsza część kodu ...

```

W ten sposób użytkownicy biblioteki muszą podać jedynie wspólną klasę nadziedną (czyli kategorię) w celu przechwycenia wszystkich wyjątków biblioteki, zarówno teraz, jak i w przyszłości.

```

# Plik client.py
import mathlib
...
try:
    mathlib.func(...)
except mathlib.NumErr:
    ...zgłoszenie i poradzenie sobie z wyjątkiem...

```

Kiedy powrócimy do kodu i zaczniemy go znowu modyfikować, nowe wyjątki możemy dodać jako nowe podklasy wspólnej klasy nadziednej.

```

# Plik mathlib.py
...
class Uflow(NumErr): pass

```

W rezultacie kod użytkownika przechwytyujący wyjątki biblioteki będzie działał *bez zmian*. Tak naprawdę w przyszłości możemy swobodnie dodawać, usuwać i modyfikować wyjątki w dowolny sposób — dopóki klient wymienia klasę nadziedną, jest izolowany od zmian w naszym zbiorze wyjątków. Innymi słowy, wyjątki oparte na klasach są o wiele lepszym rozwiążaniem pod względem utrzymywania niż łańcuchy znaków.

Hierarchie wyjątków opartych na klasach mogą także obsługiwać zachowywanie stanu oraz dziedziczenie na sposoby, które idealnie sprawdzają się w większych programach. By jednak zrozumieć te role, musimy najpierw sprawdzić, w jaki sposób klasy wyjątków zdefiniowanych przez użytkownika odnoszą się do wbudowanych wyjątków, po których dziedziczą.

Wbudowane klasy wyjątków

Tak naprawdę przykładów z poprzedniego podrozdziału nie wziąłem z powietrza. Wszystkie wbudowane wyjątki zgłasiane przez Pythona są zdefiniowane jako obiekty klas. Co więcej, są one zorganizowane w płytka hierarchię z ogólnymi klasami nadziedznymi kategorii i specyficzny typami klas podziednnych, podobnie jak wyjątki z drzewa klas, które utworzyliśmy przed chwilą.

W Pythonie 3.x wszystkie znane wyjątki, z jakimi się spotkaliśmy (na przykład `SyntaxError`), są tak naprawdę po prostu zdefiniowanymi klasami dostępnymi jako nazwy wbudowane w module `builtins` (w Pythonie 2.x znajdują się zamiast tego w module `_builtin_` i są one także atrybutami modułu biblioteki standardowej o nazwie `exceptions`). Dodatkowo Python organizuje wbudowane wyjątki w hierarchię w celu obsługiwanego różnych trybów przechwytywania. Na przykład:

BaseException: klasa na szczytce hierarchii systemowej, zawierająca domyślnego konstruktora i metody wyświetlające komunikaty

Klasa nadziedziona wyjątków najwyższego poziomu. Klasa ta nie jest przeznaczona do bezpośredniego dziedziczenia przez klasy zdefiniowane przez użytkownika (zamiast tego należy użyć `Exception`). Udostępnia domyślny sposób wyświetlania oraz zachowywanie stanu dziedziczone przez klasy podrzędne. Jeśli na instancji tej klasy wywołana zostanie wbudowana funkcja `str` (na przykład przez `print`), klasa ta zwraca łańcuch znaków wyświetlania dla argumentów konstruktora przekazanych przy tworzeniu instancji (lub pusty łańcuch znaków, jeśli argumenty były nieobecne). Dodatkowo, o ile klasa podrzędna nie zastąpi konstruktora tej klasy, wszystkie argumenty do niej przekazane w momencie tworzenia instancji przechowywane są w atrybutie `args` w postaci krotki.

Exception: klasa na szczytce hierarchii użytkownika

Klasa nadziedziona najwyższego poziomu dla wszystkich wyjątków powiązanych z aplikacjami. Jest bezpośrednią klasą podrzędną `BaseException`, a także klasą nadziedzoną wszystkich pozostałych wbudowanych wyjątków, z wyjątkiem klas zdarzeń wyjścia systemu (`SystemExit`, `KeyboardInterrupt` oraz `GeneratorExit`). Prawie wszystkie klasy zdefiniowane przez użytkownika powinny dziedziczyć po tej klasie, a nie po `BaseException`. Jeśli postępujemy zgodnie z tą konwencją, wymienienie `Exception` w programie obsługuje instrukcję `try` sprawi, że nasz skrypt będzie przechwytywał wszystko z wyjątkiem zdarzeń wyjścia systemu, które w normalnych warunkach powinny móc zostać wykonane. W rezultacie `Exception` staje się klasą przechwytyującą wszystko w instrukcjach `try` i jest bardziej dokładna od pustej części `except`.

ArithmetError: klasa na szczytce hierarchii wyjątków arytmetycznych

Klasa nadziedziona wszystkich błędów liczbowych (i jednocześnie klasa podrzędna `Exception`). Klasy podrzędne, takie jak `OverflowError`, `ZeroDivisionError` i `FloatingPointError`, reprezentują określone błędy arytmetyczne.

LookupError: klasa na szczytce hierarchii wyjątków indeksowych

Klasa pochodna od `Exception`, będąca nadklassą dla wyjątków indeksowych w sekwencjach i mapach (`IndexError` i `KeyError`), jak również dla niektórych wyjątków przeszukiwania tekstów Unicode.

Powysza lista nie jest pełna, ponieważ wbudowane wyjątki są często zmieniane i nie sposób ich wyczerpująco opisać w książce. Więcej informacji na temat tej struktury można znaleźć albo w dodatkowych materiałach, takich jak książka *Python. Leksykon kieszonkowy*, albo w dokumentacji biblioteki Pythona. Warto również zauważyć, że drzewo klas wyjątków różni się nieco w wersjach 2.x oraz 3.x szczegółami, którymi się nie będziemy tutaj zajmować, ponieważ nie są istotne.

Drzewo klas można zobaczyć w tekście pomocy modułu `exceptions` jedynie w Pythonie 2.x (więcej informacji na temat funkcji `help` można znaleźć w rozdziałach 4. oraz 15.).

```
>>> import exceptions  
>>> help(exceptions)  
...pominieto wiele tekstu...
```

Moduł ten został usunięty z wersji 3.x. Powyższe informacje można znaleźć w aktualnym systemie pomocy i innych wymienionych wyżej źródłach.

Kategorie wbudowanych wyjątków

Drzewo wbudowanych klas pozwala nam wybrać, jak bardzo specyficzny lub ogólny będzie nasz program obsługi wyjątków. Wbudowany wyjątek `ArithmeticError` jest klasą nadrzędną dla wyjątków bardziej specyficznych, takich jak `OverflowError` czy `ZeroDivisionError`.

- Wymieniając `ArithmeticError` w instrukcji `try`, przechwycimy dowolny rodzaj zgłoszonego błędu liczbowego.
- Wymieniając tylko `OverflowError`, przechwycimy jedynie ten specyficzny rodzaj błędu i żaden inny.

Podobnie, ponieważ w Pythonie 3.x `Exception` jest klasą nadrzędną wszystkich wyjątków poziomu aplikacji, możemy jej użyć w roli *klasy przechwytyjącej wszystko* — rezultat będzie przypominał puste `except`, jednak pozwala to na potraktowanie wyjątków wyjścia z systemu w sposób, jaki zazwyczaj jest pożądany.

```
try:  
    action()  
except Exception:  
    ...obsługa wszystkich wyjątków aplikacji...  
else:  
    ...obsługa przypadków bez wyjątków...
```

W Pythonie 2.x nie do końca to jednak działa, ponieważ samodzielne wyjątki zdefiniowane przez użytkownika w postaci klasycznych klas nie muszą być klasami podzielnymi klasy podstawowej `Exception`. Technika ta jest o wiele bardziej niezawodna w Pythonie 3.x, ponieważ w tej wersji wszystkie klasy muszą pochodzić od wyjątków wbudowanych. Nawet jednak w Pythonie 3.x z rozwiązaniem tym wiążą się te same, opisane w poprzednim rozdziale, potencjalne pułapki co z pustym `except` — może ono przechwytywać wyjątki przeznaczone dla innych fragmentów kodu, a także ukrywać prawdziwe błędy programistyczne. Ponieważ jest to tak często spotykany problem, powróćmy do niego w opisie pułapek w kolejnym rozdziale.

Bez względu na to, czy będziemy korzystać z kategorii we wbudowanym drzewie klas, jest to dobry przykład. Używając podobnych technik dla własnych klas wyjątków w kodzie, możemy udostępniać elastyczne oraz łatwe w modyfikacji zbiory wyjątków.



W Pythonie 3.3 została zmieniona hierarchia wyjątków wejścia/wyjścia i operacji systemowych. Pojawili się nowe klasy odpowiadające kodom błędów operacji plikowych i systemowych. Klasy tych i innych wyjątków tworzą osobną kategorię i są pochodnymi nadklassami `OSError`. Ich nazwy się nie zmieniły, aby została zachowana kompatybilność ze starszymi wersjami języka.

Wcześniej trzeba było na podstawie zawartości instancji zgłoszonego wyjątku sprawdzać, jaki konkretnie pojawił się błąd, i nierzadko zgłaszać kolejny wyjątek w celu eskalacji problemu (dla wygody programistów w module `errno` kodom błędów zostały przypisane nazwy, natomiast same kody są dostępne zarówno w podstawowej krotce `V.args[0]`, jak również w atrybucie `V.errno`). Ilustruje to poniższy kod:

```
c:\temp> py -3.2  
>>> try:
```

```
...     f = open('plik.txt')
... except IOError as V:
...     if V.errno == 2:                      # errno.N lub V.args[0]
...         print('Nie ma takiego pliku')
... else:
...     raise                                # Eskalacja innych wyjątków
...
Nie ma takiego pliku
```

Powyższy kod działa poprawnie w wersjach 3.3 i nowszych. Można jednak użyć nowych klas,ściślej odzwierciedlających wyjątki, które mogą tu wystąpić, i pomijających inne problemy:

```
c:\temp> py -3.3
>>> try:
...     f = open('plik.txt')
... except FileNotFoundError:
...     print('Nie ma takiego pliku')
...
Nie ma takiego pliku
```

Szczegółowe informacje o nowych klasach i rozszerzeniach są dostępne w wymienionych wcześniej źródłach.

Domyślne wyświetlanie oraz stan

Wbudowane wyjątki udostępniają również domyślne sposoby wyświetlania, a także zachowanie stanu. Często jest to wystarczający poziom logiki wymagany przez klasy zdefiniowane przez użytkownika. O ile nie definiujemy ponownie konstruktorów dziedziczących przez nasze klasy po wyjątkach wbudowanych, wszystkie argumenty konstruktora przekazane do tych klas zapisywane są w atrycie krotki args instancji i są automatycznie pokazywane, kiedy instancja jest wyświetlana. Jeśli nie przekazano żadnych argumentów konstruktora, używane są pusta krotka i łańcuch znaków wyświetlania.

Wyjaśnia to, dlaczego argumenty przekazane do wbudowanych klas wyjątków pokazywane są w komunikatach o błędach — wszystkie argumenty konstruktora są dołączane do instancji i pokazywane przy jej wyświetleniu:

```
>>> raise IndexError                         # To samo co IndexError() -
brak argumentów

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError

>>> raise IndexError('mielonka')            # Argument konstruktora
dołączony i wyświetlony
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: mielonka
>>> I = IndexError('mielonka')                      # Dostępny w atrybucie obiektu
>>> I.args
('mielonka',)
>>> print(I)                                         # Jawnie wyświetlony komunikat
zawiera argumenty
mielonka
```

Tak samo jest w przypadku wyjątków *zdefiniowanych przez użytkownika* w wersji Pythona 3.x (oraz w wersji 2.x w przypadku użycia składni z klasami), ponieważ dziedziczą one metody konstruktora oraz wyświetlania obecne we wbudowanych klasach nadrzędnych:

```
>>> class E(Exception): pass
...
>>> raise E
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.E
>>> raise E('spam')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.E: spam
>>> I = E('spam')
>>> I.args
('spam',)
>>> print(I)
spam
```

W instancji klasy wyjątku przechwyconego za pomocą instrukcji `try` można odwoływać się do nadziędnego konstruktora i metody wyświetlającej komunikat:

```
>>> try:
...     raise E('mielonka')
... except E as X:
...     print(X)                                         # Wyświetlenie i zapisanie
argumentów konstruktora
...     print(X.args)
...     print(repr(X))
```

```

...
mielonka
('mielonka',)
E('mielonka',)

>>> try:                                     # Zapisanie kilku argumentów w
krotce i wyświetlenie ich
... raise E('mielonka', 'jajka', 'szynka')
... except E as X:
... print('%s %s' % (X, X.args))
...
('mielonka', 'jajka', 'szynka') ('mielonka', 'jajka', 'szynka')

```

Warto zwrócić uwagę na to, że obiekty instancji wyjątków nie są łańcuchami znaków, a jedynie wykorzystują omówiony w rozdziale 30. protokół przeciążania operatorów `__str__` w celu udostępnienia łańcuchów znaków wyświetlania. W celu dokonania konkatenacji ze zwykłymi łańcuchami znaków należy wykonać ręczną konwersję: `str(X) + "łańcuch_znaków"`.

Choć automatyczna obsługa zachowywania stanu oraz sposobu wyświetlania jest przydatna sama w sobie, w przypadku specyficznych potrzeb w tym zakresie zawsze można ponownie zdefiniować odziedziczone metody, takie jak `__str__` i `__init__` w klasach podrzędnych `Exception`. W kolejnym podrozdziale pokażemy, jak to zrobić.

Własne sposoby wyświetlania

Jak widzieliśmy w poprzednim podrozdziale, domyślnie po przechwyceniu i wyświetleniu instancje wyjątków opartych na klasach wyświetlają wszystko, co przekazaliśmy do konstruktora klasy.

```

>>> class MyBad(Exception): pass
...
>>> try:
...     raise MyBad('Przepraszam -- mój błąd!')
... except MyBad as X:
...     print(X)
...
Przepraszam -- mój błąd!

```

Ten odziedziczony domyślny model wyświetlania wykorzystywany jest także wtedy, gdy wyjątek wyświetlany jest jako część komunikatu o błędzie, kiedy nie zostanie on przechwycony.

```

>>> raise MyBad('Przepraszam -- mój błąd!')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>

```

```
__main__.MyBad: Przepraszam -- mój błąd!
```

W wielu sytuacjach będzie to wystarczające. By jednak uzyskać sposób wyświetlania lepiej dostosowany do naszych potrzeb, możemy zdefiniować w klasie jedną z dwóch metod przeciążania reprezentacji łańcuchów znaków (`__repr__` lub `__str__`) w celu zwrócenia pożądanego łańcucha znaków i wyświetlenia wyjątku. Łańcuch znaków zwracany przez metodę zostanie pokazany, kiedy wyjątek zostanie albo przechwycony i wyświetlony, albo dotrze do domyślnego programu obsługi wyjątków.

```
>>> class MyBad(Exception):
...     def __str__(self):
...         return 'Zawsze patrz na życie z humorem...'
...
>>> try:
...     raise MyBad()
... except MyBad as X:
...     print(X)
...
Zawsze patrz na życie z humorem...
>>> raise MyBad()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyBad: Zawsze patrz na życie z humorem...
```

To, co zwraca metoda, zawarte zostaje w komunikatach o błędach dla wyjątków nieprzechwyconych, a także wykorzystane zostaje przy wyświetlaniu wyjątków w jawnym sposób. W powyższym kodzie w celach ilustracyjnych metoda zwraca zapisany na stałe łańcuch znaków, jednak może ona także wykonywać dowolne przetwarzanie tekstu, na przykład z wykorzystaniem informacji o stanie dołączanych do obiektu instancji. Opcjom związanym z informacjami o stanie poświęcony jest następny podrozdział.



Drobna uwaga, którą warto uwzględnić: zazwyczaj w tym celu należy zdefiniować ponownie metodę `__str__`, ponieważ wbudowane klasy nadzędne wyjątków ją zawierają, a metoda ta jest bardziej pożądana niż `__repr__` w większości kontekstów (w tym w trakcie wyświetlania). Jeśli zdefiniujemy `__repr__`, w trakcie wyświetlania wywołana zostanie zamiast tego metoda `__str__` klasy nadzędnej!

```
>>> class E(Exception):
...     def __repr__(self): return 'Metoda niewywołana!'
...
>>> raise E('spam')
...
__main__.E: spam
>>> class E(Exception):
...     def __str__(self): return 'Metoda wywołana!'
```

```
>>> raise E('spam')
...
__main__.E: Metoda wywołana!
Więcej informacji na temat tych metod specjalnych można znaleźć w rozdziale 30.
```

Właśnie dane oraz zachowania

Poza obsługą elastycznych hierarchii klasy wyjątków udostępniają również miejsce na dodatkowe informacje o stanie w postaci atrybutów instancji. Jak widzieliśmy wcześniej, wbudowane klasy nadrzędne wyjątków udostępniają domyślny konstruktor automatycznie zapisujący argumenty konstruktora w atrybucie krotki instancji o nazwie `args`. Choć konstruktor domyślny w wielu przypadkach będzie wystarczający, w niektórych sytuacjach możemy udostępnić własny. Dodatkowo klasy mogą definiować metody wykorzystywane w programach obsługi, udostępniające gotową logikę przetwarzania wyjątków.

Udostępnianie szczegółów wyjątku

Kiedy wyjątek jest zgłoszany, może on przechodzić granice plików — wywołującą wyjątek instrukcja `raise` i przechwytyująca go instrukcja `try` mogą się znajdować w zupełnie innych plikach modułów. Zazwyczaj nie jest najlepszym pomysłem przechowywanie dodatkowych szczegółów w zmiennych globalnych, ponieważ instrukcja `try` może nie wiedzieć, w którym pliku znajdują się te zmienne. Przekazywanie dodatkowych informacji o stanie w samym wyjątku daje instrukcji `try` bardziej niezawodny dostęp do nich.

W przypadku klas odbywa się to niemalże automatycznie. Jak widzieliśmy, kiedy zgłoszany jest wyjątek, Python przekazuje obiekt instancji klasy wraz z wyjątkiem. Kod z instrukcji `try` może uzyskać dostęp do zgłoszonej instancji, wymieniając dodatkowe zmienne po słowie kluczowym `as` w programie obsługi `except`. Daje nam to naturalny punkt zaczepienia dla dostarczania danych oraz zachowania do programu obsługi.

Przykładowo program analizujący pliki danych może sygnalizować błąd składniowy, zgłaszając instancję wyjątku wypełnioną dodatkowymi szczegółami dotyczącymi błędu.

```
>>> class FormatError(Exception):
...     def __init__(self, line, file):
...         self.line = line
...         self.file = file
...
...
>>> def parser():
...     raise FormatError(42, file='spam.txt')      # Kiedy znaleziony zostanie
błąd
...
...
>>> try:
...     parser()
```

```

... except FormatError as X:
...     print('Błąd w', X.file, X.line)
...
Błąd w spam.txt 42

```

W części `except` zmiennej `X` jest przypisywana referencja do instancji wygenerowanej, kiedy zgłoszony zostanie wyjątek. Daje nam to dostęp do atrybutów dołączonych do instancji przez własny konstruktor. Choć moglibyśmy polegać na obsłudze zachowanego stanu we wbudowanych klasach nadzędnych, dla naszej aplikacji ma to mniejsze znaczenie (poza tym nie można byłoby umieszczać ciągów znaków w argumentach metody, jak w poprzednim przykładzie).

```

>>> class FormatError(Exception): pass           # Odziedziczony konstruktor
...
>>> def parser():
...     raise FormatError(42, 'spam.txt')          # Słowa kluczowe nie są
dozwolone!
...
>>> try:
...     parser()
... except FormatError as X:
...     print('Błąd w:', X.args[0], X.args[1])    # Nie jest specyficzne dla
tej aplikacji
...
Błąd w: 42 spam.txt

```

Udostępnianie metod wyjątków

Poza umożliwieniem obsługi specyficznych dla aplikacji informacji o stanie własne konstruktory lepiej obsługują dodatkowe zachowania obiektów wyjątków. Klasa wyjątku może również definiować *metody* wywoływanie w programie obsługi. Poniższy kod (plik `excparse.py`) dodaje na przykład metodę wykorzystującą informacje o stanie wyjątku do zapisania błędów w pliku.

```

from __future__ import print_function      # Zachowanie kompatybilności z
wersją 2.x

class FormatError(Exception):
    logfile = 'formaterror.txt'

    def __init__(self, line, file):
        self.line = line
        self.file = file

    def logerror(self):
        log = open(self.logfile, 'a')
        print('Błąd w', self.file, self.line, file=log)

```

```

def parser():
    raise FormatError(40, 'spam.txt')
if __name__ == '__main__':
    try:
        parser()
    except FormatError as exc:
        exc.logerror()

```

Po wykonaniu powyższy skrypt zapisuje komunikat o błędzie do pliku w odpowiedzi na wywołanie metody w programie obsługi wyjątku:

```

c:\code> del formaterror.txt
c:\code> py -3 excparse.py
c:\code> py -2 excparse.py
c:\code> type formaterror.txt
Błąd w spam.txt 40
Błąd w spam.txt 40

```

W takiej klasie metody (jak `logerror`) mogą również być odziedziczone po klasach nadrzędnych, a atrybuty instancji (jak `line` oraz `file`) udostępniają miejsce do zapisywania informacji o stanie udostępniających dodatkowy kontekst do wykorzystania w późniejszych wywołaniach metod. Co więcej, klasy wyjątków mogą w dowolny sposób rozszerzać odziedziczone zachowania i dostosowywać je do własnych potrzeb.

```

class CustomFormatError(FormatError):
    def logerror(self):
        ...własny, unikatowy kod...
    raise CustomFormatError(...)

```

Innymi słowy, ponieważ wyjątki są w Pythonie definiowane za pomocą klas, wszystkie omówione w szóstej części książki zalety programowania zorientowanego obiektowo dostępne są do użycia.

Na koniec dwie uwagi: po pierwsze obiekt wyjątku przypisany w powyższym kodzie zmiennej `exc` jest również drugim elementem krotki zwrocionej przez metodę `sys.exc_info()`. Metoda ta zawiera informacje o ostatnio zgłoszonym wyjątku. Korzysta się z niej wtedy, gdy nie używa się nazwy klasy po instrukcji `except`, ale potrzebny jest dostęp do informacji o stanie wyjątku i do jego metod. Po drugie metoda `logerror()` dopisuje do pliku dziennika niestandardowy komunikat, ale może też generować standardowy komunikat o błędzie wraz ze śladem stosu, wykorzystując narzędzia i obiekty dostępne w bibliotece `traceback`.

Więcej informacji o metodzie `sys.exc_info()` i śladzie stosu zawartych jest w następnym rozdziale.

Podsumowanie rozdziału

W niniejszym rozdziale zajęliśmy się tworzeniem wyjątków zdefiniowanych przez użytkownika. Jak się dowiedzieliśmy, wyjątki implementowane są jako obiekty instancji klas w Pythonie 2.6 oraz 3.0 (wcześniej alternatywa w postaci modelu wyjątków opartych na łańcuchach znaków dostępna była w starszych wersjach, jednak obecnie jest ona przestarzała). Klasy wyjątków obsługują koncepcję hierarchii wyjątków (ułatwiającą późniejsze utrzymanie kodu), pozwalając na dołączanie danych oraz zachowania do wyjątków w postaci atrybutów i metod instancji, a także pozwalają na dziedziczenie w instancjach danych oraz zachowania po klasach nadzędnych.

Widzieliśmy, że w instrukcji `try` przechwycenie klasy nadzędnej przechwytuje tę klasę, a także wszystkie klasy podrzędne znajdujące się pod nią w drzewie klas. Klasy nadzędne stają się nazwami kategorii wyjątków, a klasy podrzędne stają się bardziej specyficznymi typami wyjątków wewnątrz tych kategorii. Widzieliśmy również, że wbudowane klasy nadzędne wyjątków, po których musimy dziedziczyć, udostępniają przydatne opcje domyślne wyświetlania oraz zachowywania stanu. W miarę potrzeby możemy je jednak nadpisać.

W kolejnym rozdziale zakończymy tę część książki, omawiając pewne często spotykane przypadki użycia wyjątków oraz badając narzędzia wykorzystywane często przez programistów Pythona. Zanim jednak tam przejdziemy, czas na quiz podsumowujący niniejszy rozdział.

Sprawdź swoją wiedzę – quiz

1. Jakie są dwa nowe ograniczenia, którym w Pythonie 3.x podlegają wyjątki zdefiniowane przez użytkownika?
2. W jaki sposób zgłasiane wyjątki oparte na klasach dopasowywane są do programów obsługi?
3. Należy wymienić dwa sposoby pozwalające dołączyć informacje kontekstowe do obiektów wyjątków.
4. Należy wymienić dwa sposoby pozwalające na podanie tekstu komunikatu o błędzie dla obiektów wyjątków.
5. Dlaczego nie powinniśmy już dzisiaj używać wyjątków opartych na łańcuchach znaków?

Sprawdź swoją wiedzę – odpowiedzi

1. W Pythonie 3.x wyjątki muszą być definiowane za pomocą klas (oznacza to, że zgłoszany i przechwytywany jest obiekt instancji klasy). Dodatkowo klasy wyjątków muszą pochodzić od wbudowanej klasy `BaseException` (większość programów dziedziczy po jej klasie podrzędnej `Exception`, by móc obsługiwać programy wszystkich błędów dla normalnych typów wyjątków).
2. Wyjątki oparte na klasach dopasowywane są po związkach z klasami nadzędnymi. Podanie klasy nadzędnej w programie obsługuje przechwyci instancje tej klasy, a także instancje wszystkich jej klas podrzędnych znajdujących się niżej w drzewie klas. Z tego powodu możemy sobie wyobrazić klasy nadzędne jako ogólne kategorie wyjątków, a klasy podrzędne jako bardziej szczegółowe typy wyjątków wewnątrz tych kategorii.

3. Informacje kontekstowe możemy dołączać do wyjątków opartych na klasach, wypełniając atrybuty instancji w zgłoszonym obiekcie instancji, zazwyczaj we własnym konstruktorze klasy. W przypadku mniej skomplikowanych potrzeb wbudowane klasy nadrzędne wyjątków udostępniają konstruktor automatycznie przechowujący argumenty w instancji (w atrybucie `args`). W programach obsługuje wyjątków wymienia się zmienną, która ma być przypisana do zgłoszonej instancji, a następnie przechodzi jej nazwę w celu uzyskania dostępu do dołączonych informacji o stanie i wywołania metod zdefiniowanych w klasie.
4. Tekst komunikatu o błędzie w wyjątkach opartych na klasach można określić za pomocą własnej metody przeciążania operatorów `__str__`. W przypadku mniej skomplikowanych potrzeb wbudowane klasy nadrzędne wyjątków automatycznie wyświetlały wszystkie informacje przekazane do konstruktora klasy. Operacje takie, jak `print` czy `str` automatycznie pobierająła串 znaków wyświetlania obiektu wyjątku, kiedy jest on pokazywany, albo w sposób bezpośredni, albo jako część komunikatu o błędzie.
5. Ponieważ tak powiedział Guido — zostały one usunięte w wersjach 2.6 oraz 3.0 Pythona. A tak naprawdę istnieją ku temu dobre powody. Wyjątki oparte na łańcuchach znaków nie obsługiwały kategorii, informacji o stanie ani dziedziczenia zachowania w sposób, w jaki robią to wyjątki oparte na klasach. W praktyce sprawiało to, że z wyjątków opartych na klasach łatwiej korzystało się na początku, kiedy programy były małe, ale coraz trudniej było to robić w miarę ich rozrastania się.

Wyjątki jako klasy mają ten mankament, że w celu obsłużenia ich w istniejącym kodzie trzeba go gruntownie zmienić. Ponadto początkujący programista, który chce definiować nowe wyjątki, czy też generalnie dogłębnie je poznać, musi zawsze przyswoić sobie szeroką wiedzę, tj. poznać klasy i programowanie obiektowe. Z tego właśnie powodu pozornie prosty temat został poruszony dopiero w tej części książki. Niemniej jednak tego rodzaju zależności pomiędzy różnymi zagadnieniami są typowe dla obecnych wersji języka Python (szerzej temat był opisany we wstępie i wnioskach).

[1] Jak zasugerował jeden mój mądry student, moduł biblioteki mógłby również udostępniać obiekt krotki zawierający wszystkie wyjątki zgłaszane przez bibliotekę. Klient importowałby wtedy krotkę i podawał ją w części `except` w celu przechwycenia wszystkich wyjątków biblioteki (przypomnijmy, że krotka w `except` oznacza *dowolny* z jej wyjątków). Kiedy później dodamy nowy wyjątek, biblioteka musi jedynie rozszerzyć eksportowaną krotkę. Takie rozwiązanie działa, jednak nadal musielibyśmy aktualnić krotkę zgodnie z wyjątkami zgłaszanymi w module biblioteki. Wyjątki oparte na klasach oferują również więcej zalet niż tylko kategorie — obsługują też dziedziczenie stanu oraz metod, a także model dostosowywania do własnych potrzeb — coś, czego nie robią proste, pojedyncze wyjątki.

Rozdział 36. Projektowanie z wykorzystaniem wyjątków

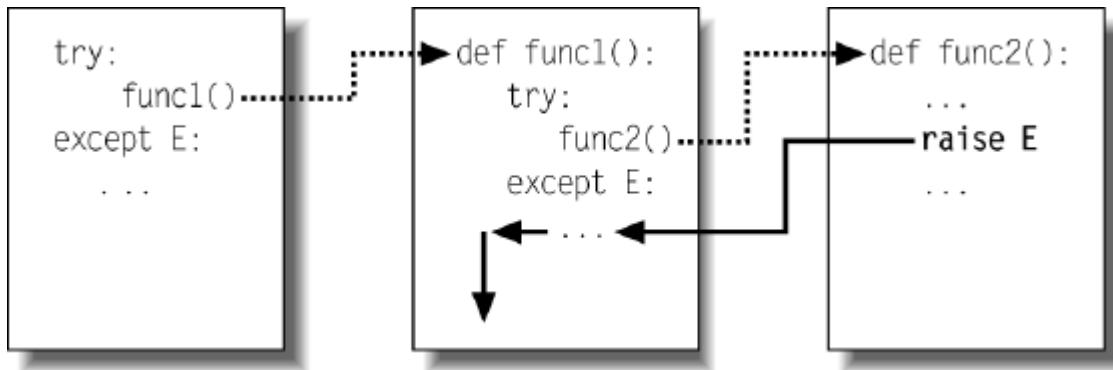
Niniejszy rozdział kończy tę część książki kolekcją zagadnień związanych z projektowaniem wyjątków, a także przykładami często spotykanych przypadków użycia, po których zamieszczono omówienie pułapek oraz ćwiczenia przeznaczone dla tej części książki. Ponieważ rozdział ten kończy również główną część książki, zawiera krótkie omówienie narzędzi programistycznych, które mogą nam pomóc przejść z poziomu początkującego użytkownika Pythona do poziomu programisty aplikacji w tym języku.

Zagnieżdżanie programów obsługi wyjątków

Nasze dotychczasowe przykłady wykorzystywały pojedynczą instrukcję `try` do przechwytywania wyjątków, jednak co się stanie, jeśli jedna instrukcja `try` zostanie zagnieżdżona wewnętrz innej? I co oznacza, jeśli `try` wywołuje funkcję, która wykonuje kolejną instrukcję `try`? Z technicznego punktu widzenia instrukcje `try` mogą być zagnieżdżane w kategoriach składni oraz przebiegu sterowania w programie w czasie wykonywania. Wspomniałem już o tym wcześniej, ale teraz przyjrzyjmy się dokładniej temu zagadnieniu.

Oba te przypadki możemy zrozumieć, jeśli zdamy sobie sprawę z tego, że Python w czasie wykonywania układu instrukcje `try na stosie`. Kiedy zgłoszany jest wyjątek, Python wraca do ostatniej instrukcji `try` z pasującą częścią `except`. Ponieważ każda instrukcja `try` pozostawia znacznik, Python może przeskoczyć do wcześniejszych `try`, badając ułożone na stosie znaczniki. To właśnie zagnieżdżanie aktywnych programów obsługi mamy na myśli, kiedy mówimy o przekazywaniu (propagacji) wyjątków do „wyższych” programów obsługi — takie programy obsługi to po prostu instrukcje `try`, do których wesliśmy *wcześniej* w czasie przebiegu wykonywania programu.

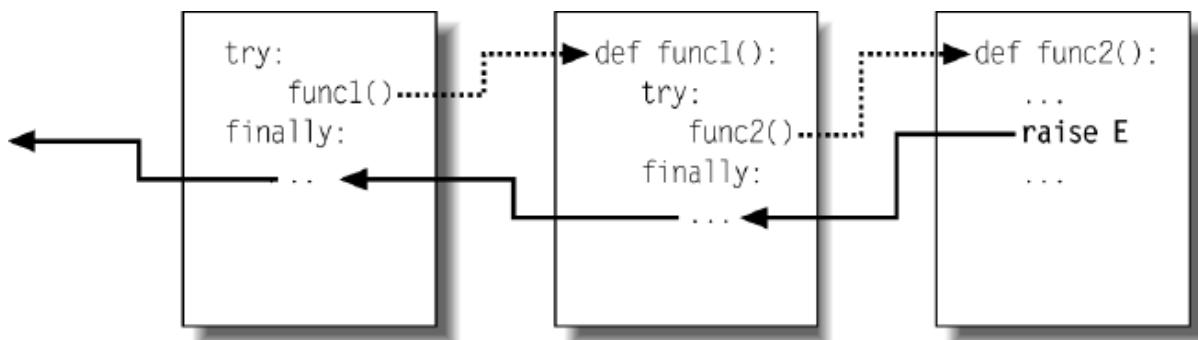
Na rysunku 36.1 przedstawiono, co się dzieje, kiedy instrukcje `try` z częściami `except` zagnieżdżane są w czasie wykonywania. Ilość kodu, jaka trafia do bloku `try`, może być dość znaczna i może zawierać wywołania funkcji wywołujące inny kod, który może śledzić te same wyjątki. Kiedy wyjątek zostanie w końcu zgłoszony, Python przeskakuje z powrotem do ostatniej instrukcji `try` wymieniającej ten wyjątek, wykonuje część `except` tej instrukcji, a następnie wznowia wykonywanie po niej.



Rysunek 36.1. Zagnieżdżone instrukcje try/except. Kiedy wyjątek zostanie zgłoszony (przez nas bądź przez Pythona), sterowanie przeskakuje z powrotem do ostatniej odwiedzonej instrukcji try z pasującą częścią except, a program jest wznawiany po tej instrukcji try. Części except przechwytyują i zatrzymują wyjątek — to w nich przetwarza się wyjątki oraz radzi sobie z nimi

Po przechwyceniu wyjątku jego życie dobiera końca — sterowanie nie przeskakuje z powrotem do *wszystkich* pasujących instrukcji try wymieniających ten wyjątek, ponieważ jedynie pierwsza z nich otrzymuje szansę na obsłużenie wyjątku. Na rysunku 36.1 widać na przykład, że instrukcja raise z funkcji func2 przesyła sterowanie z powrotem do programu obsługi z funkcji func1, a następnie program jest kontynuowany w func1.

W przeciwieństwie do powyższej sytuacji, kiedy zagnieżdżane są instrukcje try zawierające jedynie części finally, w momencie wystąpienia wyjątku wykonywany jest każdy blok finally po kolei. Python kontynuuje przekazywanie wyjątku do góry do innych instrukcji try, a na końcu być może nawet do domyślnego programu obsługi najwyższego poziomu (wyświetlającego standardowe komunikaty o błędach). Jak widać na rysunku 36.2, części finally nie kończą wyjątku — po prostu określają one kod, jaki ma być wykonany przy okazji wychodzenia z każdej instrukcji try w czasie procesu przekazywania wyjątku. Jeśli wiele części finally z instrukcji try jest aktywnych w czasie wystąpienia wyjątku, wykonane zostaną *wszystkie*, o ile część except nie przechwyci wyjątku gdzieś po drodze.



Rysunek 36.2. Zagnieżdżone instrukcje try/finally. Kiedy wyjątek zostanie zgłoszony w tej sytuacji, sterowanie powraca do ostatniej instrukcji try w celu wykonania jej części finally, a następnie wyjątek przekazywany jest do wszystkich finally we wszystkich aktywnych instrukcjach try, aż wreszcie dociera do domyślnego programu obsługi na najwyższym poziomie, gdzie wyświetlany jest komunikat o błędzie. Części finally przechwytyują wyjątek (jednak go nie zatrzymują) — służą do zamieszczania działań, które mają być wykonywane przy wyjściu

Innymi słowy, to, dokąd podąża program po zgłoszeniu wyjątku, jest całkowicie uzależnione od tego, *gdzie był* — jest to funkcją przebiegu sterowania w skrypcie w czasie wykonywania, a nie

tylko kwestią jego składni. Przekazywanie wyjątku w gruncie rzeczy przechodzi z powrotem do instrukcji `try`, do których weszliśmy, ale których jeszcze nie opuściliśmy. Przekazywanie zatrzymuje się, gdy tylko sterowanie odnajduje pasującą część `except`, jednak nie kiedy przechodzi przez części `finally` znajdujące się po drodze.

Przykład – zagnieżdżanie przebiegu sterowania

Przejdzmy teraz do przykładu, by skonkretyzować nieco koncepcje związane z takim zagnieżdżaniem. Poniższy plik modułu `nestexc.py` definiuje dwie funkcje. Funkcja `action2` została zapisana w taki sposób, by wywoływać wyjątek (nie da się dodawać liczb i sekwencji), natomiast funkcja `action1` opakowuje wywołanie `action2` w program obsługi `try` w celu przechwycenia wyjątku.

```
def action2():
    print(1 + [])
    # Wygenerowanie wyjątku
TypeError

def action1():
    try:
        action2()
    except TypeError:                      # Najbardziej aktualna pasująca
        instrukcja try
        print('wewnętrzne try')

    try:
        action1()
    except TypeError:                      # Tutaj tylko jeśli action1
        ponownie zgłasza wyjątek
        print('zewnętrzne try')

% python nestexc.py
wewnętrzne try
```

Warto jednak zauważyć, że kod najwyższego poziomu znajdujący się na dole pliku opakowuje wywołanie `action1` w kolejny program obsługi `try`. Kiedy funkcja `action2` wywołuje wyjątek `TypeError`, dwie instrukcje `try` będą aktywne — ta z funkcji `action1` oraz ta znajdująca się na najwyższym poziomie pliku. Python wybiera i wykonuje ostatnią instrukcję `try` z pasującą częścią `except`, którą w tym przypadku okazuje się instrukcja `try` znajdująca się wewnątrz funkcji `action1`.

Jak wspomniałem, miejsce, do którego przeskakuje wyjątek, uzależnione jest od przebiegu sterowania w programie w czasie wykonywania. Z tego powodu, by wiedzieć, gdzie trafiemy, musimy wiedzieć, gdzie jesteśmy. W tym przypadku to, gdzie wyjątki są obsługiwane, jest raczej funkcją przebiegu sterowania, a nie składni instrukcji. Możemy jednak również zagnieżdżać programy obsługi wyjątków za pomocą składni, co zobaczymy poniżej.

Przykład – zagnieżdżanie składniowe

Jak wspomniałem przy okazji omawiania połączonej instrukcji `try/except/finally` w rozdziale 34., można również zagnieżdżać instrukcje `try` składniowo, przez ich pozycję w kodzie źródłowym.

```

try:
    try:
        action2()
    except TypeError: # Najbardziej aktualna pasująca
instrukcja try
        print('wewnętrzne try')

except TypeError: # Tutaj tylko jeśli action1
    ponownie zgłasza wyjątek
    print('zewnętrzne try')

```

Tak naprawdę kod ten ustawia tylko tę samą strukturę zagnieżdżonych programów obsługi wyjątków co poprzedni przykład (zachowuje się również tak samo). Zagnieżdżanie składniowe działa dokładnie tak samo jak przypadki z rysunków 36.1 oraz 36.2. Jedyna różnica polega na tym, że zagnieżdżone programy obsługi są fizycznie osadzone wewnątrz bloku `try`, a nie umieszczone w funkcjach znajdujących się gdzie indziej. Przykładowo zagnieżdżone części `finally` wszystkie wykonywane są w momencie wystąpienia wyjątku, bez względu na to, czy są one zagnieżdżone składniowo, czy za pomocą przebiegu wykonywania w fizycznie oddzielonych od siebie częściach kodu.

```

>>> try:
...     try:
...         raise IndexError
...     finally:
...         print('mielonka')
... finally:
...     print('MIELONKA')
...
mielonka
MIELONKA
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    IndexError

```

Graficzna ilustracja działania tego kodu znajduje się na rysunku 36.2. Rezultat jest taki sam, jednak logika funkcji została wstawiona tutaj jako instrukcje zagnieżdżone. Bardziej użyteczny przykład działania zagnieżdżania składniowego można znaleźć w poniższym pliku `except-finally.py`.

```

def raise1(): raise IndexError
def noraise(): return
def raise2(): raise SyntaxError
for func in (raise1, noraise, raise2):
    print('<%s>' % func.__name__)

```

```
try:  
    try:  
        func()  
    except IndexError:  
        print('przechwycono IndexError')  
finally:  
    print('wykonano finally')  
print('...')
```

Kod ten przechwytuje wyjątek (jeśli zostanie on zgłoszony) i wykonuje działania końcowe części `finally` bez względu na ewentualne wystąpienie wyjątku. Zrozumienie tego kodu może zajść chwilę, jednak jego rezultat jest w dużej mierze taki sam jak dzisiejsze połączenie `except` oraz `finally` w jednej instrukcji `try` w Pythonie 2.5 i późniejszych wersjach.

```
% python except-finally.py  
<raise1>  
przechwycono IndexError  
wykonano finally  
<noraise>  
wykonano finally  
<raise2>  
wykonano finally  
Traceback (most recent call last):  
  File "except-finally.py", line 9, in <module>  
    func()  
  File "except-finally.py", line 3, in raise2  
    def raise2(): raise SyntaxError  
SyntaxError: None
```

Jak widzieliśmy w rozdziale 34., od Pythona 2.5 części `except` oraz `finally` można łączyć w tej samej instrukcji `try`. Dodatkowo możliwość stosowania wielu instrukcji `except` sprawia, że niektóre opisane tutaj techniki zagnieżdżania składniowego nie są potrzebne. Są one jednak wciąż stosowane, szczególnie w większych programach i kodach napisanych w wersjach Pythona starszych niż 2.5. Role instrukcji `except` i `finally` są w nich wyraźniej rozgraniczone i można je traktować jako alternatywne sposoby implementowania obsługi wyjątków.

Zastosowanie wyjątków

Zapoznaliśmy się już z mechanizmami będącymi podstawą wyjątków. Przyjrzymy się teraz niektórym sposobom typowego wykorzystania wyjątków.

Wychodzenie z głęboko zagnieżdzonych pętli: instrukcja go to

Jak wspomniano na początku książki, wyjątki czasami mogą pełnić rolę instrukcji `go to` stosowanej w innych językach, umożliwiającej przechodzenie do dowolnej części programu. Za pomocą wyjątków można to robić w sposób uwzględniający strukturę kodu, precyzujący miejsce w zagnieżdzonych blokach, do którego następuje przejście.

Pod tym względem instrukcja `raise` jest podobna do `go to`, a instrukcje `except` z nazwami wyjątków można porównać do etykiet. Przechodzić można jednak tylko do miejsc kodu objętego instrukcją `try`. Jest to bardzo ważne, ponieważ program wykorzystujący instrukcję `go to`, która pozwala przechodzić do całkowicie dowolnych miejsc, może być z tego powodu wyjątkowo nieczytelny i trudny w utrzymaniu.

W Pythonie instrukcja `break` powoduje wyjście tylko z bieżącej pętli, natomiast zgłaszając wyjątek, można w razie potrzeby wyjść poza kilka zagnieżdzonych pętli:

```
>>> class Exitloop(Exception): pass  
  
...  
>>> try:  
...     while True:  
...         while True:  
...             for i in range(10):  
...                 if i > 3: raise Exitloop      # Instrukcja break spowodowałaby  
wyjście tylko jeden poziom wyżej  
...                 print('pętla 3: %s' % i)  
...                 print('pętla 2')  
...                 print('pętla 1')  
... except Exitloop:  
...     print('kontynuacja')            # Ew. instrukcja pass  
kontynuująca działanie kodu  
  
...  
pętla 3: 0  
pętla 3: 1  
pętla 3: 2  
pętla 3: 3  
kontynuacja  
>>> i  
4
```

Jeżeli instrukcję `raise` zmieni się na `break`, wtedy pętla będzie wykonywana w nieskończoność, ponieważ instrukcja ta będzie przerwała tylko najbardziej zagnieżdzoną pętlę, po czym kontynuowane będzie wykonywanie kolejnej pętli w hierarchii zagnieżdżeń. Zostanie wyświetlony napis „pętla 2”, po czym proces rozpocznie się od nowa.

Należy zwrócić uwagę, że zmienna i wciąż istnieje po zakończeniu wykonywania bloku `try`. Generalnie zmienne użyte wewnętrz bloku `try` zachowują przypisane im wartości, czym różnią się od zmiennych, którym za pomocą instrukcji `except` są przypisywane instancje klas wyjątków, oraz zmiennych stosowanych w funkcjach. Takie zmienne są traktowane jako lokalne i tracone po zgłoszeniu wyjątku. Z technicznego punktu widzenia zmienne te są zdejmowane ze stosu, a obiekty, do których się odwołują, są likwidowane podczas oczyszczania pamięci. Proces ten przebiega automatycznie.

Wyjątki nie zawsze są błędami

W Pythonie wszystkie błędy są wyjątkami, ale nie wszystkie wyjątki są błędami. W rozdziale 9. widzieliśmy na przykład, że metody odczytu plików zwracają pusty łańcuch znaków na końcu pliku. Wbudowana funkcja `input` (przedstawiona najpierw w rozdziale 3., a później zastosowana w pętli interaktywnej w rozdziale 10.) wczytuje wiersz tekstu ze standardowego strumienia wejścia (`sys.stdin`) w każdym wywołaniu i zgłasza wbudowany `EOFError` na końcu pliku. Funkcja ta w Pythonie 2.x nosi nazwę `raw_input`.

W przeciwieństwie do metod plików funkcja ta nie zwraca pustego łańcucha znaków — pusty łańcuch znaków z `input` oznacza pusty wiersz. Pomimo nazwy wyjątek `EOFError` (czyli ang. *end-of-file error*, błąd końca pliku) jest w tym kontekście tylko sygnałem, a nie błędem. Ze względu na to zachowanie, o ile koniec pliku nie powinien kończyć skryptu, funkcja `input` często pojawia się opakowana w program obsługi wyjątków `try` i zagnieżdzona w pętli, jak w poniższym kodzie.

```
while True:  
    try:  
        line = input()                      # Wczytanie wiersza ze stdin (w wersji  
        # 2.x jest to funkcja raw_input)  
        except EOFError:  
            break                           # Wyjście z pętli na końcu pliku  
        else:  
            ...przetwarzanie kolejnego wiersza...
```

Kilka innych wbudowanych wyjątków w podobny sposób jest sygnałami, a nie błędami. Przykładowo wywołanie `sys.exit()` i naciśnięcie na klawiaturze przycisków `Ctrl+C` powodują, odpowiednio, zgłoszenie wyjątków `SystemExit` oraz `KeyboardInterrupt`.

Python zawiera również zbiór wbudowanych wyjątków reprezentujących raczej *ostrzeżenia* niż błędy. Niektóre z nich służą do sygnalizowania przestarzałych opcji języka. Informacje o wbudowanych wyjątkach można znaleźć w dokumentacji biblioteki standardowej, natomiast dokumentacja modułu `warnings` zawiera więcej szczegółów dotyczących ostrzeżeń.

Funkcje mogą sygnalizować warunki za pomocą `raise`

Wyjątki zdefiniowane przez użytkownika mogą również sygnalizować warunki niebędące błędami. Procedura wyszukiwania może na przykład zostać utworzona tak, by zgłaszać wyjątek, kiedy dopasowanie zostanie odnalezione, zamiast zwracać opcję statusu, którą wywołujący musi zinterpretować. W poniższym kodzie program obsługuje wyjątków `try/except/else` wykonuje pracę testu sprawdzającego zwracaną wartość `if/else`.

```
class Found(Exception): pass  
def searcher():
```

```

if ...sukces...:
    raise Found()                      # Zgłoszenie wyjątku zamiast zwrotienia
flagi
else:
    return

try:
    searcher()

except Found:                         # Wyjątek, jeśli element został
odnaleziony
    ...sukces...
else:                                  # else zwracane, kiedy element nie został
odnaleziony
    ...porażka...

```

Mówiąc bardziej ogólnie, taka struktura kodu może przydać się każdej funkcji, która nie może zwrócić wartości w celu oznaczenia sukcesu bądź porażki. Jeśli na przykład wszystkie obiekty są potencjalnie poprawnymi zwracanymi wartościami, żadna z tych wartości nie będzie w stanie zasygnalizować niezwykłych warunków. Wyjątki umożliwiają sygnalizowanie wyników bez zwracania wartości.

```

class Failure(Exception): pass

def searcher():
    if ...sukces...:
        return ...znaleziony element...
    else:
        raise Failure()

try:
    item = searcher()
except Failure:
    ...zgłoszenie...
else:
    ...tutaj wykorzystanie elementu...

```

Ponieważ Python jest językiem z typami dynamicznymi oraz obsługą polimorfizmu, wyjątki (a nie zwracane wartości) są na ogół preferowanym sposobem sygnalizowania takich warunków.

Zamykanie plików oraz połączeń z serwerem

Przykłady z tej kategorii spotkaliśmy w rozdziale 34. W skrócie, narzędzia do przetwarzania wyjątków wykorzystywane są często do zakończenia korzystania z zasobów systemowych, bez względu na to, czy w trakcie przetwarzania wystąpił błąd.

Przykładowo niektóre serwery wymagają zamykania połączeń w celu zakończenia sesji. W podobny sposób pliki wyjścia mogą wymagać zamknięcia wszystkich wywołań w celu zrzucenia

buforów na dysk, natomiast pliki wejścia bez zamknięcia mogą zużywać deskryptory plików. Choć obiekty plików są zamykane automatycznie, kiedy pamięć jest czyszczona — w trakcie gdy są one jeszcze otwarte — czasami trudno jest przewidzieć, kiedy to nastąpi.

Najbardziej ogólnym i bezpośrednim sposobem gwarantującym działania końcowe określonego bloku kodu jest instrukcja `try/finally`.

```
myfile = open(r'C:\misc\script', 'w')

try:
    ...przetworzenie myfile...

finally:
    myfile.close()
```

Jak widzieliśmy w rozdziale 34., niektóre obiekty ułatwiają to w Pythonie 2.6 oraz 3.0, udostępniając *menedżery kontekstu* wykonywane przez instrukcję `with/as` i zamykające dla nas obiekt automatycznie.

```
with open(r'C:\misc\script', 'w') as myfile:
    ...przetworzenie myfile...
```

Która opcja będzie zatem lepszym rozwiązaniem? Jak zwykle zależy to od programu. W porównaniu z `try/finally` menedżery kontekstu są *bardziej niejawne*, co jest sprzeczne z ogólną filozofią projektową Pythona. Menedżery kontekstu są także bez wątpienia *mniej ogólne* — dostępne są jedynie dla wybranych obiektów, a pisanie zdefiniowanych przez użytkownika menedżerów kontekstu obsługujących ogólne wymagania w zakresie zamykania jest bardziej skomplikowane od użycia instrukcji `try/finally`.

Z drugiej strony, użycie istniejących menedżerów kontekstu wymaga *mniejszej ilości kodu* niż w przypadku skorzystania z `try/finally`, co widać w powyższych przykładach. Co więcej, protokół menedżerów kontekstu obok działań końcowych (wyjścia) obsługuje także działania *początkowe* (wejścia). W praktyce, jeżeli nie są spodziewane żadne wyjątki, można zaoszczędzić kilka wierszy kodu (kosztem jednak późniejszego zagnieźdzenia i wcięcia właściwego bloku):

```
myfile = open(filename, 'w')      # Forma tradycyjna
...przetwarzanie pliku...

myfile.close()

with open(filename) as myfile: # Forma z użyciem menedżera kontekstu
    ...przetwarzanie pliku...
```

Niejawną obsługę wyjątków za pomocą instrukcji `with` można porównać do jawnej obsługi za pomocą instrukcji `try/finally`. Choć `try/finally` jest chyba techniką o szerszym zastosowaniu, menedżery kontekstu mogą być bardziej odpowiednim wyborem, kiedy są już dostępne albo gdy ich dodatkowy poziom skomplikowania ma uzasadnienie.

Debugowanie z wykorzystaniem zewnętrznych instrukcji `try`

Programy obsługują również wykorzystać do zastąpienia domyślnego zachowania programów obsługi błędów Pythona znajdujących się na najwyższym poziomie. Opakowując cały program (lub jego wywołanie) w zewnętrzną instrukcję `try` w kodzie najwyższego poziomu, możemy przechwytywać wszystkie wyjątki, jakie mogą wystąpić w czasie wykonywania tego programu, tym samym odwracając domyślne zakończenie programu.

W poniższym kodzie pusta część `except` przechwytuje wszystkie nieprzechwycone wyjątki zgłoszone w trakcie wykonywania programu. By uzyskać prawdziwy wyjątek, jaki wystąpił, należy pobrać wynik wywołania funkcji `sys.exc_info` z wbudowanego modułu `sys`. Funkcja ta zwraca krotkę, której dwa pierwsze elementy zawierają klasę bieżącego wyjątku, a także zgłoszony obiekt instancji (więcej informacji na temat funkcji `sys.exc_info` za moment).

try:

 ...wykonanie programu...

except:

 wyjątki trafiają tutaj

 # Wszystkie nieprzechwycone

 import sys

 print('nie przechwycono!', sys.exc_info()[0], sys.exc_info()[1])

Struktura ta jest często wykorzystywana w czasie programowania w celu zachowania działania programów nawet po wystąpieniu błędów — pozwala ona wykonać dodatkowe testy bez konieczności ponownego uruchamiania. Jest ona również stosowana przy testowaniu kodu innego programu, co opisane jest w kolejnym podrozdziale.



Więcej informacji o obsługuiwaniu wyjątków *bez kontynuowania* wykonywania kodu jest dostępnych w dokumentacji do standardowej biblioteki `atexit`. Za pomocą tej biblioteki można dostosowywać operacje wykonywane przez funkcję `sys.excepthook` wykorzystywaną przez proces obsługi wyjątków na najwyższym poziomie. Oprócz tego w powyższej dokumentacji opisane są inne narzędzia do podobnych zastosowań.

Testowanie kodu wewnętrz tego samego procesu

Niektóre z omówionych wyżej wzorców kodu można połączyć w aplikacji testującej inny kod wewnętrz tego samego procesu. Ilustruje to ogólnie poniższy szablon kodu:

```
import sys
log = open('testlog', 'a')
from testapi import moreTests, runNextTest, testName
def testdriver():
    while moreTests():
        try:
            runNextTest()
        except:
            print('PORĄŻKA', testName(), sys.exc_info()[:2], file=log)
        else:
            print('SUKCES', testName(), file=log)
testdriver()
```

W powyższym kodzie funkcja `testdriver` przechodzi serię wywołań sprawdzających (szczególny modułu `testapi` pominiemy). Ponieważ nieprzechwycony wyjątek w testowanym kodzie spowodowałby zakończenie całego procesu funkcji przeprowadzającej test, musimy opakować

wywołania testowanego kodu w instrukcję `try`, jeśli chcemy móc kontynuować proces testowania po jego niepowodzeniu. Jak zawsze pusta część `except` przechwytyuje wszystkie nieprzechwycone wyjątki wygenerowane przez testowany kod i wykorzystuje funkcję `sys.exc_info` do zapisania wyjątków do pliku. Część `else` wykonywana jest wtedy, gdy nie wystąpią żadne wyjątki, czyli kiedy test się powiedzie.

Taki ustandaryzowany kod jest typowy dla systemów sprawdzających funkcje, moduły oraz klasy za pomocą wykonania ich w tym samym procesie co kod wykonujący testy. W praktyce jednak testowanie może być o wiele bardziej wyszukane od powyższego przykładu. By na przykład testować *programy zewnętrzne*, moglibyśmy zamiast tego sprawdzać kody statusu lub dane wyjściowe generowane przez narzędzia uruchamiające programy, takie jak `os.system` oraz `os.popen`, omówione w dokumentacji biblioteki standardowej (takie narzędzia na ogół nie zgłaszą wyjątków dla błędów w programach zewnętrznych — tak naprawdę testowany kod może działać równolegle do kodu go testującego).

Na końcu rozdziału spotkamy również kilka późniejszych platform testowych udostępnianych przez Pythona, takich jak `doctest` oraz `PyUnit`, które oferują narzędzia służące do porównywania oczekiwanych danych wyjściowych z prawdziwymi wynikami.

Więcej informacji na temat funkcji `sys.exc_info`

Wynik funkcji `sys.exc_info` wykorzystany w dwóch poprzednich przykładach pozwala programowi obsługi wyjątków na uzyskanie dostępu do ostatnio zgłoszonego wyjątku. Jest to szczególnie przydatne, kiedy wykorzystujemy pustą część `except` w celu ślepego przechwycenia wszystkich wyjątków i ustalenia, co zostało zgłoszone.

```
try:  
    ...  
except:  
    # sys.exc_info()[0:2] to klasa i instancja wyjątku
```

Jeśli żaden wyjątek nie jest obsługiwany, wywołanie to zwraca krotkę zawierającą trzy wartości `None`. W przeciwnym razie zwracane wartości to krotka (`typ, wartość, ślad`), gdzie:

- `typ` to klasa obsługiwanej wyjątku,
- `wartość` to instancja klasy wyjątku, która została zgłoszona,
- `ślad` to obiekt śladu reprezentujący stos wywołań w momencie, w którym początkowo wystąpił wyjątek, a moduł `traceback` wygenerował komunikat o błędzie.

Jak widzieliśmy w poprzednim rozdziale, `sys.exc_info` może się czasami przydać do ustalenia określonego typu wyjątku przy przechwytywaniu klas nadzędnych kategorii wyjątków. Ponieważ jednak, jak widzieliśmy, w tym przypadku możemy także uzyskać typ wyjątku, pobierając atrybut `_class_` instancji uzyskany za pomocą części `as`, `sys.exc_info` jest obecnie wykorzystywany przede wszystkim przez pustą część `except`.

```
try:  
    ...  
except General as instance:  
    # instance._class_ to klasa wyjątku
```

Jeżeli w powyższym kodzie w miejscu identyfikatora `General` użyje się nazwy `Exception`, wtedy będzie można przechwytywać wszystkie wyjątki. Jest to konstrukcja podobna do pustej instrukcji `except`, ale mniej nietypowa, dająca dostęp do instancji klasy wyjątku. Skorzystanie z interfejsów obiektu instancji oraz polimorfizmu jest zatem często lepszym rozwiązaniem od

sprawdzania typów wyjątków — metody wyjątków można definiować dla poszczególnych klas i wykonywać w sposób ogólny:

```
try:  
    ...  
except General as instance:  
    # instance.method() robi, co należy, dla tej instancji
```

Jak zwykle bycie zbyt specyficzny w Pythonie może ograniczyć elastyczność kodu. Rozwiązanie polimorficzne, takie jak w ostatnim przykładzie, lepiej obsługuje przyszłą ewolucję kodu.

Wyświetlanie błędów i śladów stosu

Obiekt zwracany przez opisaną w poprzednim podrozdziale funkcję `sys.exc_info` można wykorzystywać do jawnego generowania komunikatów o błędach i śladów stosu za pomocą standardowego modułu `traceback`. Moduł ten oferuje kilka interfejsów, za pomocą których można go elastycznie dostosowywać do własnych potrzeb. W niniejszej książce nie ma miejsca na pełny opis tego tematu, ale podstawowe informacje są proste. Przeanalizujmy poniższy kod (plik `badly.py`):

```
import traceback  
  
def inverse(x):  
    return 1 / x  
  
try:  
    inverse(0)  
except Exception:  
    traceback.print_exc(file=open('badly.exc', 'w'))  
    print('Koniec')
```

Zastosowana została tutaj przydatna funkcja `print_exc` dostępna w module `traceback`, domyślnie wykorzystująca informacje zwarcane przez funkcję `sys.exc_info`. Powyższy kod zapisuje w pliku komunikat o błędzie. Jest to rozwiązanie szczególnie przydane podczas testowania programu, gdy trzeba przechwytywać wyjątki i rejestrować wszystkie związane z nimi informacje. Ilustruje to poniższy kod:

```
c:\code> python badly.py  
Koniec  
c:\code> type badly.exc  
Traceback (most recent call last):  
  File "badly.py", line 7, in <module>  
    inverse(0)  
  File "badly.py", line 4, in inverse  
    return 1 / x  
ZeroDivisionError: division by zero
```

Znacznie więcej informacji o obiektach śledzących wyjątki, module traceback i pokrewnych tematach jest dostępnych w odpowiednich podręcznikach i innych źródłach.



Uwaga na temat wersji: W Pythonie 2.x starsze narzędzia sys.exc_type oraz sys.exc_value nadal działają i służą do pobierania typu oraz wartości wyjątku, jednak mogą zarządzać tylko jednym, globalnym wyjątkiem na cały proces. Te dwie nazwy zostały usunięte w Pythonie 3.x. Nowsze i preferowane wywołanie sys.exc_info() dostępne w wersji 2.x oraz 3.x śledzi natomiast informacje o wyjątkach dla każdego wątku, dlatego jest powiązane z wątkiem. Oczywiście rozróżnienie to ma znaczenie tylko wtedy, gdy w programach napisanych w Pythonie wykorzystuje się wątki (zagadnienie to wykracza poza zakres niniejszej książki, jednak Python 3.x wymusza tę kwestię). Więcej szczegółów znajduje się w bardziej zaawansowanych tekstuach.

Wskazówki i pułapki dotyczące projektowania wyjątków

W tym rozdziale łączę ze sobą wskazówki projektowe oraz pułapki, ponieważ okazuje się, że najczęściej spotykane pułapki w dużej mierze wynikają z kwestii związanych z projektowaniem. Wyjątki są generalnie w Pythonie łatwe w użyciu. Prawdziwą sztuką jest jednak zdecydowanie, jak szczegółowe lub ogólne mają być nasze części except oraz jak dużo kodu należy opakować w instrukcje try. Spróbujmy najpierw odnieść się do tego drugiego problemu.

Co powinniśmy opakować w try

Moglibyśmy opakować każdą instrukcję skryptu w osobną instrukcję try, jednak byłoby to po prostu głupie (instrukcje try musiałyby następnie być opakowywane w następne instrukcje try!). Tak naprawdę jest to zagadnienie dotyczące projektowania, wykraczające poza sam język, które stanie się bardziej oczywiste z czasem i nabieranym doświadczeniem. Poniżej znajduje się jednak kilka ogólnych reguł, które mogą się przydać już teraz.

- Operacje, które często kończą się niepowodzeniem, powinny być opakowywane w instrukcje try. Pierwszorzędnymi kandydatami do instrukcji try są na przykład operacje wchodzące w interakcje ze stanem systemu (jak otwieranie plików czy wywołania gniazd).
- Istnieją jednak wyjątki od powyższej reguły — w prostym skrypcie możemy chcieć, by niepowodzenie takiej operacji kończyło nasz program zamiast przechytywania i ignorowania błędów. Jest tak szczególnie wtedy, gdy to niepowodzenie jest prawdziwą przeszkołą dla działania programu. Niepowodzenie w Pythonie zazwyczaj kończy się użytecznym komunikatem o błędzie (a nie całkowitym wyłączeniem systemu), dlatego często jest to najlepszy wynik, na jaki mogliśmy liczyć.
- Powinniśmy implementować działania końcowe w instrukcjach try/finally, by zagwarantować ich wykonanie, o ile niedostępny jest menedżer kontekstu w postaci opcji with/as. Ta forma instrukcji pozwala na wykonywanie kodu w dowolnych scenariuszach bez względu na to, czy wystąpią wyjątki.
- Czasami wygodniej jest opakować wywołanie większej funkcji w pojedynczą instrukcję try, niż zaśmiecać samą funkcję licznymi instrukcjami try. W ten sposób wszystkie wyjątki z funkcji są przekazywane do instrukcji try znajdującej się wokół wywołania i redukujemy ilość kodu wewnętrz funkcji.

Typy programów, jakie będziemy pisać, najprawdopodobniej wpłyną na ilość obsługi wyjątków stosowaną przez nas w kodzie. Serwery muszą działać stale, dlatego będą na ogół wymagały

instrukcji `try` przechwytyujących wyjątki i pomagających sobie z nimi radzić. Programy *testujące* inny kod w jednym procesie, jakie widzieliśmy w tym rozdziale, najprawdopodobniej również będą obsługiwały wyjątki. Prostsze, często jednorazowe skrypty często będą całkowicie ignorowały obsługę wyjątków, ponieważ niepowodzenie na dowolnym etapie skryptu będzie powodowało jego zakończenie.

Jak nie przechwytywać zbyt wiele – unikanie pustych except i wyjątków

Przejdźmy do kwestii ogólności wyjątków. Python pozwala nam wybrać, które wyjątki chcemy przechwytywać, jednak czasami musimy uważać, by nie obejmować zbyt wiele. Widzieliśmy już na przykład, że pusta część `except` przechwytuje *każdy* wyjątek, jaki może zostać zgłoszony, kiedy wykonywany jest kod z bloku `try`.

Rozwiążanie to łatwo jest utworzyć w kodzie i czasami jest ono pożądane, jednak może się okazać, że przechwycimy w ten sposób błąd, który jest oczekiwany przez program obsługi try gdzieś wyżej w strukturze zagnieźdżenia wyjątków. Przykładowo program obsługi wyjątków taki, jak poniższy przechwytuje i zatrzymuje każdy wyjątek, jaki do niego dociera, bez względu na to, czy nie oczekuje go inny program obsługi.

```
def func():
    try:
        ...
    except IndexError:
        ...
    except:
        ...
    ...
    się zatrzymuje
try:
    func()
except IndexError:
    # Wyjątek powinien być
    # przetworzony tutaj
```

Co gorsza, taki kod może również przechwycić niezwiązane wyjątki systemowe. W Pythonie wyjątki zgłaszą nawet takie zdarzenia, jak błędy pamięci, prawdziwe błędy programistyczne, zatrzymania iteracji, przerwania z klawiatury czy wyjścia z systemu. Takie wyjątki zazwyczaj nie powinny być przechwytywane, chyba że piszemy debugera lub inne podobne narzędzie.

Przykładowo skrypty normalnie kończą działanie, kiedy sterowanie wyjdzie poza koniec pliku najwyższego poziomu. Python udostępnia jednak również wywołanie `sys.exit(statuscode)` pozwalające na wczesne zakończenie. Takie rozwiązanie działa, zgłaszając wbudowany wyjątek `SystemExit` w celu zakończenia programu. Jeśli zatem jakieś programy `try/finally` działają przy wychodzeniu, specjalne typy programów mogą przechwycić to zdarzenie^[1]. Z tego powodu instrukcja `try` z pustą częścią `except` może w niezamierzony sposób zapobiec kluczowemu zakończeniu działania, jak w poniższym pliku `exiter.py`.

```
import sys  
  
def bye():  
    sys.exit(40) # Kluczowy błąd – zakończenie!
```

```

try:
    bye()
except:
    print('mam go')                                # Oj – zignorowaliśmy wyjście
    print('kontynuuję...')

% python exiter.py
mam go
kontynuuję...

```

Tak naprawdę nie jesteśmy w stanie przewidzieć wszystkich rodzajów wyjątków, jakie mogą wystąpić w czasie operacji. Użycie wbudowanych klas wyjątków z poprzedniego rozdziału może nam pomóc w tym akurat przypadku, ponieważ klasa nadziedzona `Exception` nie jest klasą nadziedzoną `SystemExit`:

```

try:
    bye()
except Exception:                               # Nie przechwyci wyjąć, ale
    *przechwyci* wiele innych wyjątków
    ...

```

W innych przypadkach takie rozwiązanie nie jest lepsze od pustej części `except`. Ponieważ `Exception` jest klasą nadziedzoną wszystkich wbudowanych wyjątków poza zdarzeniami wyjścia z systemu, nadal może potencjalnie przechwycić wyjątki przeznaczone dla innego miejsca programu.

Co chyba najgorsze, zarówno pusta część `except`, jak i przechwytywanie klasy `Exception` przechwytyują również prawdziwe błędy programistyczne, które powinny przez większość czasu być dozwolone. Tak naprawdę te dwie techniki mogą w rezultacie *wyłączyć* mechanizm zgłoszania błędów Pythona, co sprawi, że zauważenie błędów w kodzie stanie się naprawdę trudne. Rozważmy na przykład poniższy kod.

```

mydictionay = {...}

...
try:
    x = mydictionay['mielonka']                # Oj – błąd w pisowni
except:
    x = None                                     # Zakłada, że mamy KeyError
    ...kontynuujemy z x...

```

Powyższy kod zakłada, że jedynym błędem, jaki może nam się przytrafić w trakcie indeksowania słownika, jest błąd brakującego klucza. Jednak ponieważ nazwa `mydictionay` jest napisana z błędem (powinno to być `mydictionay`), Python zgłasza wyjątek `NameError` zamiast referencji do niezdefiniowanej zmiennej. Błąd ten zostanie po cichu przechwycony przez program obsługi wyjątków i zignorowany. Program obsługi niepoprawnie wypełni wartość domyślną w dostępie do słownika, maskując błąd programu.

Co więcej, przechwycenie `Exception` nic tu nie da, ponieważ rezultat byłby taki sam, jaki dałoby użycie pustej części `except`: szczerliwe wykonanie po cichu domyślnego kodu, maskujące prawdziwą przyczynę błędu, którą na pewno warto było znać. Jeśli takie coś

zdarzy się w kodzie znacznie oddalonym od miejsca, w którym wykorzystane są pobrane wartości, usunięcie takiego błędu może być dużym wyzwaniem!

Z reguły należy w programach obsługi wyjątków być tak *specyficzny*, jak to możliwe — puste części `except` i przechwytywanie `Exception` są poręczne, ale potencjalnie bardzo podatne na błędy. W ostatnim przykładzie lepiej byłoby na przykład napisać `except KeyError:`, by nasze intencje były jasne i by uniknąć przechwytywania niezwiązań zdarzeń. W prostszych skryptach potencjalna możliwość wystąpienia problemów może nie być na tyle znacząca, by przeważyć nad wygodą stosowania pustych części `except` przechwytyjących wszystko, jednak najczęściej ogólne programy obsługi oznaczają kłopoty.

Jak nie przechwytywać zbyt mało — korzystanie z kategorii opartych na klasach

Z drugiej strony, nasze programy obsługi wyjątków nie powinny być nadmiernie specyficzne. Kiedy w instrukcji `try` wymienimy określone wyjątki, przechwycimy jedynie to, co jest na tej liście. Nie zawsze jest to złe rozwiązanie, ale jeśli system ewoluje i może w przyszłości zwracać inne wyjątki, być może trzeba będzie powrócić do kodu i dodać te wyjątki do list znajdujących się w każdym miejscu.

Z tym zjawiskiem spotkaliśmy się w poprzednim rozdziale. Poniższy program obsługi został napisany w taki sposób, by traktować `MyExcept1` oraz `MyExcept2` jako normalne przypadki, a wszystko inne jako błąd. Tym samym jeśli w przyszłości dodamy również przypadek `MyExcept3`, zostanie on przetworzony jako błąd, o ile nie uaktualnimy listy wyjątków.

```
try:  
    ...  
  
    except (MyExcept1, MyExcept2):          # Przestaje działać po dodaniu  
        MyExcept3  
        ...                                # Nie-błędy  
  
    else:                                  # Zakładamy, że jest błędem  
        ...
```

Na szczęście rozważne użycie omówionych w rozdziale 34. wyjątków opartych na klasach może sprawić, że pułapka ta zupełnie zniknie. Jak widzieliśmy, jeśli przechwycimy ogólną klasę nadziedną, możemy w przyszłości dodawać bardziej specyficzne klasy podzielone bez konieczności ręcznego rozszerzania listy z części `except` — klasa nadziedna staje się rozszerzalną kategorią wyjątków.

```
try:  
    ...  
  
    except SuccessCategoryName:            # Będzie OK, jeśli dodamy  
        podklasę MyExcept3  
        ...                                # Nie-błędy  
  
    else:                                  # Zakładamy, że jest błędem  
        ...
```

Innymi słowy, niewielka ilość projektowania ma spore konsekwencje. Morał z tej historii jest taki, że należy uważać, by w programach obsługi wyjątków nie być ani zbyt ogólnym, ani zbyt specyficznym, i mądrze wybierać poziom szczegółowości naszych instrukcji `try` opakowujących

kod. W szczególności w przypadku większych systemów polityka dotycząca wyjątków powinna być częścią całościowego projektu.

Podsumowanie podstaw języka Python

Gratulacje! Niniejszy tekst kończy nasze omówienie podstaw języka Python. Każdy, kto dotarł aż do tej strony, może się od teraz uznawać za Oficjalnego Programistę Pythona. Kolejnym krokiem jest lektura bardziej zaawansowanych materiałów, które wymienię za chwilę. Na tym jednak kończy się opis podstaw języka Python, co jest głównym celem tej książki.

Widzieliśmy już w zasadzie wszystko, co jest do zobaczenia w samym języku, a do tego nieraz na poziomie wystarczająco zaawansowanym, aby większość opisanych technik móc stosować w praktyce. Omówiliśmy typy wbudowane, instrukcje oraz wyjątki, a także narzędzia wykorzystywane do budowania większych jednostek programów (takich, jak funkcje, moduły oraz klasy). Zapoznaliśmy się nawet z najważniejszymi zagadnieniami, takimi jak projektowanie programów, programowanie obiektowe i funkcyjne, architektura programów, zalety i wady różnych narzędzi. Wszystko razem stanowi wiedzę, którą z całą mocą można wykorzystywać do tworzenia aplikacji z prawdziwego zdarzenia.

Zbiór narzędzi Pythona

Od teraz nasza przyszła kariera programisty Pythona będzie w dużej mierze zależała od doskonalenia się w wykorzystywaniu *zbioru narzędzi* dostępnych dla programowania na poziomie aplikacji. Jak się okaże, jest to zajęcie ciągłe. Biblioteka standardowa zawiera na przykład setki modułów, a w internecie można znaleźć jeszcze więcej narzędzi. Można spokojnie spędzić całą dekadę, próbując podnieść swoje umiejętności w zakresie wykorzystywania tych narzędzi, w szczególności dlatego, że ciągle pojawiają się nowe (wiem, co mówię!).

Mówiąc ogólnie, Python udostępnia następującą hierarchię zbiorów narzędzi:

Narzędzia wbudowane

Typy wbudowane, takie jak łańcuchy znaków, listy czy słowniki, sprawiają, że proste programy pisze się o wiele szybciej.

Rozszerzenia Pythona

Dla bardziej wymagających zadań możemy rozszerzyć Pythona, pisząc własne funkcje, moduły oraz klasy.

Skompilowane rozszerzenia

Choć nie omawialiśmy tego zagadnienia w książce, Pythona można również rozszerzać za pomocą modułów napisanych w innych zewnętrznych językach programowania, takich jak C czy C++.

Ponieważ Python układa swoje zbiory narzędzi w warstwy, możemy zadecydować, jak bardzo nasze programy mają się zagłębiać w tę hierarchię dla określonego zadania — narzędzia wbudowane możemy wykorzystywać w prostych skryptach, na potrzeby większych systemów dodawać rozszerzenia napisane w Pythonie, natomiast do zadań zaawansowanych zastosować skompilowane rozszerzenia. W książce omówiliśmy jedynie pierwsze dwie kategorie, jednak powinno to w zupełności wystarczyć do rozpoczęcia całkiem rozsądniego programowania w Pythonie.

Oprócz tego są narzędzia, zasoby i techniki, dzięki którym język Python można stosować niemal w każdej dziedzinie informatyki. Wskazówki, w której stronie można teraz pójść, znajdują się w

rozdziale 1. zawierającym przegląd zastosowań Pythona. Wkrótce okaże się, że w prostym języku programowania, jakim jest Python, często wykonywane zadania są o wiele prostsze, niż można się tego spodziewać.

Narzędzia programistyczne przeznaczone do większych projektów

Większość przykładów opisanych w tej książce to były małe, samodzielne programy, specjalnie napisane tak, aby pomagały poznać podstawy języka. Jednak teraz, gdy wiemy o rdzeniu języka wszystko, czas nauczyć się stosować w praktyce wbudowane i zewnętrzne interfejsy.

W rzeczywistości programy pisane w Pythonie są znacznie większe od przykładów, na których eksperymentowaliśmy w tej książce. Nietrywialne, praktyczne aplikacje, wykorzystujące wszystkie potrzebne moduły, nierzadko składają się z tysięcy wierszy. Choć podstawowe narzędzia strukturalne, takie jak moduły i klasy, pozwalają zapanować nad złożonością kodu, warto czasami wspomóc się dodatkowymi rozwiązaniami.

Dla celów rozwijania większych systemów w samym Pythonie oraz w internecie istnieje zbiór narzędzi programistycznych. Działanie niektórych z nich już widzieliśmy, o innych tylko wspomniałem. Poniżej znajduje się krótkie omówienie narzędzi najczęściej wykorzystywanych w tej dziedzinie.

PyDoc oraz łańcuchy znaków dokumentacji

Funkcja `help` z PyDoc oraz interfejsy HTML zostały omówione w rozdziale 15. PyDoc udostępnia system dokumentacji przeznaczony dla modułów oraz obiektów i integruje się z łańcuchami znaków dokumentacji Pythona. Więcej informacji o tworzeniu dokumentacji kodu źródłowego jest zawartych w rozdziałach 15. i 4.

PyChecker oraz PyLint

Ponieważ Python jest językiem tak dynamicznym, niektóre błędy programistyczne nie są zgłasiane, dopóki program nie będzie wykonywany (błędy składni są na przykład przechwytywane, kiedy plik jest wykonywany lub importowany). Nie jest to wielka wada — tak jak w większości języków programowania, oznacza to tylko tyle, że musimy przetestować kod przed udostępnieniem go. W najgorszym razie w przypadku Pythona zastępujemy fazę komplikacji fazą początkowych testów. Co więcej, dynamiczna natura Pythona, automatyczne komunikaty o błędach oraz model wyjątków sprawiają, że łatwiej i szybciej jest znaleźć i naprawić błędy w Pythonie niż w innych językach programowania (w przeciwieństwie do języka C, Python nie przestaje działać w momencie wystąpienia błędu).

Niemniej jednak warto skorzystać z innych narzędzi. Systemy PyChecker oraz PyLint udostępniają obsługę przechwytywania dużego zbioru często spotykanych błędów z wyprzedzeniem, zanim skrypt zostanie wykonany. Pełnią podobną rolę jak program `lint` w programowaniu w języku C. Niektóre grupy programistów Pythona przepuszczają swój kod przez system PyChecker przed testowaniem lub publikowaniem w celu przechwycenia wszystkich potencjalnych problemów. Tak naprawdę biblioteka standardowa Pythona jest regularnie sprawdzana przez PyChecker przed opublikowaniem. PyChecker i PyLint to pakiety zewnętrzne na licencji open source. Można je znaleźć na stronie <http://www.python.org>, w witrynie PyPI lub za pomocą wyszukiwarki.

PyUnit (inaczej unittest)

W rozdziale 25. widzieliśmy, w jaki sposób do pliku Pythona dodaje się kod samosprawdzający, wykorzystując sztuczkę `__name__ == '__main__'` umieszczoną na dole pliku. Dla bardziej zaawansowanych celów testowania Python zawiera dwa narzędzia wspomagające sprawdzanie kodu. Pierwsze z nich, PyUnit (w dokumentacji biblioteki nazywane `unittest`), udostępnia zorientowaną obiektywnie platformę klas służącą do

określania oraz dostosowywania do własnych potrzeb przypadków testowych oraz oczekiwanych rezultatów. Naśladuje ona platformę JUnit przeznaczoną dla języka Java. Jest to wyszukiwany system testów jednostkowych oparty na klasach; więcej informacji na jego temat można znaleźć w dokumentacji biblioteki Pythona.

doctest

Moduł biblioteki standardowej *doctest* udostępnia drugie i prostsze podejście do testów regresyjnych. Oparty jest na łańcuchach znaków dokumentacji Pythona. By użyć *doctest*, należy w przybliżeniu wyciąć i wkleić log testowej sesji interaktywnej do łańcuchów znaków w plikach źródłowych. Moduł *doctest* dokonuje ekstrakcji łańcuchów znaków dokumentacji, pobiera z nich przypadki testowe oraz wyniki i wykonuje testy kolejny raz w celu zweryfikowania oczekiwanych wyników. Operacje *doctest* można na różne sposoby dostosowywać do własnych potrzeb; więcej informacji na ten temat można znaleźć w dokumentacji biblioteki.

Zintegrowane środowiska programistyczne

Zintegrowane środowiska programistyczne (IDE) przeznaczone dla Pythona omawialiśmy w rozdziale 3. IDE takie, jak IDLE udostępniają środowisko graficzne służące do edycji, wykonywania, debugowania oraz przeglądania naszych programów napisanych w Pythonie. Niektóre zaawansowane środowiska programistyczne (takie jak Eclipse, Komodo, NetBeans i inne wymienione w rozdziale 3.) mogą również obsługiwać dodatkowe zadania programistyczne, w tym integrację z systemami kontroli wersji, refaktoryzację kodu czy narzędzia do zarządzania projektami. Więcej informacji na temat dostępnych zintegrowanych środowisk programistycznych oraz narzędzi do budowania graficznych interfejsów użytkownika można znaleźć w rozdziale 3., na stronie poświęconej edytorom tekstu w witrynie <http://www.python.org> oraz za pomocą wyszukiwarki internetowej.

Programy profilujące

Ponieważ Python jest językiem wysokopoziomowym oraz dynamicznym, kwestie związane z wydajnością poznane dzięki doświadczeniu z innymi językami programowania zazwyczaj nie mają zastosowania do kodu napisanego w tym języku. By naprawdę wyizolować „wąskie gardła” dla wydajności kodu, musimy za pomocą narzędzi zegarowych dodać logikę pomiarową dostępną w modułach *time* oraz *timeit* lub wykonać swój kod pod modułem *profile*. Przykład działania modułów pomiarowych widzieliśmy przy okazji porównywania szybkości narzędzi iteracyjnych w rozdziale 21.

Profilowanie jest zazwyczaj pierwszym krokiem ku optymalizacji kodu — kod jest profilowany w celu wyizolowania „wąskich gardeł”, a następnie dokonuje się pomiarów dla alternatywnych rozwiązań. Moduł *profile* pochodzi z biblioteki standardowej i implementuje program profilujący dla kodu źródłowego napisanego w Pythonie. Wykonuje on dostarczony łańcuch kodu (na przykład operację importowania pliku skryptu lub wywołanie funkcji), a następnie, domyślnie, wyświetla w standardowym strumieniu wyjścia raport podający statystyki wydajności — liczbę wywołań każdej funkcji, czas spędzony w każdej funkcji i dużo więcej.

Moduł *profile* można wykonać w postaci skryptu lub zimportować. Można go także na różne sposoby dostosowywać do własnych potrzeb. Może on na przykład zapisywać statystyki wykonywania do pliku, który będzie później analizowany za pomocą modułu *pstats*. W celu dokonania interaktywnego profilowania należy zimportować moduł *profile* i wywołać *profile.run('kod')*, przekazując kod, który chcemy sprofilować, w postaci łańcucha znaków (na przykład wywołanie funkcji, zimportowanie całego pliku). W celu wykonania profilowania z systemowego wiersza poleceń powłoki należy skorzystać z polecenia *python -m profile main.py args...* (więcej informacji na temat tego formatu znajduje się w dodatku A). Inne opcje profilowania omówione są w dokumentacji biblioteki standardowej Pythona. Moduł *cProfile* ma na przykład identyczne interfejsy jak *profile*, jednak jego wykonanie jest mniej kosztowne, przez co może się lepiej nadawać do zastosowania w przypadku długiego działających programów.

Debugery

W rozdziale 3. omówiliśmy także opcje debugowania kodu (ramka „Debugowanie kodu w Pythonie”). Słownem przypomnienia, większość środowisk programistycznych przeznaczonych dla Pythona obsługuje debugowanie oparte na graficznym interfejsie użytkownika, a biblioteka standardowa Pythona udostępnia również moduł debugowania kodu źródłowego o nazwie `pdb`. Moduł ten udostępnia interfejs wiersza poleceń i działa w przybliżeniu jak popularne debugery języka C (na przykład `dbx`, `gdb`).

Podobnie jak narzędzie do profilowania, debugger `pdb` możemy wykorzystywać albo interaktywnie, albo z wiersza poleceń; można go także zimportować i wywołać z programu napisanego w Pythonie. By użyć go w sposób interaktywny, importujemy moduł, zaczynamy wykonywać kod, wywołując funkcję `pdb` (na przykład `pdb.run("main()")`), a następnie wpisujemy polecenia debugowania z interaktywnej zachęty `pdb`. W celu uruchomienia `pdb` z systemowego wiersza poleceń powłoki należy użyć polecenia `python -m pdb main.py args....` Moduł `pdb` zawiera również przydatne narzędzie do analizy post mortem, funkcję `pdm.pm()`, które włącza debugger po napotkaniu wyjątku. Wymagane jest przy tym użycie polecenia `python` z opcją `-i`. Więcej informacji na ten temat jest dostępnych w dodatku A.

Ponieważ zintegrowane środowiska programistyczne takie jak IDLE zawierają interfejsy do debugowania za pomocą klikania, moduł `pdb` nie jest dzisiaj narzędziem krytycznym, z wyjątkiem sytuacji, gdy GUI nie jest dostępne albo pożądany jest wyższy stopień kontroli. Więcej wskazówek na temat wykorzystywania graficznego interfejsu debugera w IDLE można znaleźć w rozdziale 3. Tak naprawdę w praktyce ani `pdb`, ani debugery ze zintegrowanych środowisk programistycznych nie są zbyt często wykorzystywane. Jak pisaliśmy w rozdziale 3., większość programistów albo wstawia po prostu do kodu instrukcje `print`, albo czyta komunikaty o błędach Pythona (nie jest to może najbardziej nowoczesne rozwiązanie, ale jego praktyczność decyduje o jego przewadze w świecie Pythona).

Opcje udostępniania kodu

W rozdziale 2. wprowadziliśmy narzędzia służące do pakowania programów Pythona. `PyInstaller`, `py2exe` oraz `freeze` mogą pakować kod bajtowy oraz maszynę wirtualną Pythona w zamrożone pliki binarne będące samodzielnymi plikami wykonywalnymi niewymagającymi zainstalowania Pythona na komputerze docelowym i w pełni ukrywającymi kod systemu. Dodatkowo w rozdziale 2. książki widzieliśmy, że programy napisane w Pythonie mogą być udostępniane w postaci źródeł (`.py`) lub kodu bajtowego (`.pyc`), a punkty zaczepienia operacji importowania obsługują specjalne techniki pakowania, takie jak automatyczna ekstrakcja plików `.zip` oraz szyfrowanie kodu bajtowego.

Przelotnie zapoznaliśmy się z modułami biblioteki standardowej `distutils`, które udostępniają opcje pakowania modułów oraz pakietów Pythona, a także rozszerzeń w języku C. Więcej informacji na ten temat można znaleźć w dokumentacji Pythona. Nowszy zewnętrzny system pakowania o nazwie „eggs” udostępnia kolejną alternatywę powiązaną z zależnościami; więcej informacji na jego temat można znaleźć w internecie.

Opcje optymalizacji

Istnieje kilka możliwości optymalizacji programów. Opisany w rozdziale 2. system PyPy udostępnia kompilator JIT służący do przekładania kodu bajtowego Pythona na binarny kod maszynowy, natomiast Shedskin oferuje translator z Pythona na język C++. Czasami można się również spotkać ze zoptymalizowanymi plikami z kodem bajtowym o rozszerzeniu `.pyo`, które są generowane oraz wykonywane z opcją wiersza poleceń `-O` (omówioną w rozdziałach 22. oraz 34. i zastosowaną w rozdziale 39.). Ponieważ daje to jednak naprawdę skromną przewagę z zakresie wydajności, nie jest wykorzystywane zbyt często, zazwyczaj do usuwania z programu kodu diagnostycznego.

Jako ostatnią deskę ratunku można potraktować przeniesienie części programu do języka skompilowanego, takiego jak C, w celu podniesienia ich wydajności. Więcej informacji na temat rozszerzeń języka C można znaleźć w książce *Programming Python*, a także dokumentacji Pythona. Mówiąc ogólnie, szybkość Pythona poprawia się z czasem, dlatego kiedy tylko jest to możliwe, należy uaktualniać Pythona do szybszej wersji. (Warto jednak sprawdzać, czy program faktycznie działa szybciej. Pierwotna wersja Pythona 3.0, później wielokrotnie poprawiana, wykonywała niektóre operacje wejścia/wyjścia nawet tysiąc razy wolniej niż wersja 2.x!).

Inne wskazówki przeznaczone dla większych projektów

W tekście spotkaliśmy wreszcie wiele różnych opcji języka, które stają się bardziej przydatne w miarę tworzenia większych projektów. Pośród nich są między innymi pakiety modułów (rozdział 24.), wyjątki oparte na klasach (rozdział 34.), atrybuty pseudoprivatne klas (rozdział 31.), łańcuchy znaków dokumentacji (rozdział 15.), pliki konfiguracyjne ścieżki modułów (rozdział 22.), ukrywanie zmiennych przed instrukcjami `from *` za pomocą list `__all__` oraz nazw w stylu `_X` (rozdział 25.), dodawanie kodu samosprawdzającego za pomocą sztuczki `__name__ == '__main__'` (rozdział 25.), wykorzystywanie często stosowanych reguł projektowania dla funkcji oraz modułów (rozdziały 17., 19. oraz 25.) czy wykorzystanie zorientowanych obiektywnie wzorców programowania (rozdział 31. i inne).

By dowiedzieć się więcej na temat dostępnych w internecie narzędzi programowania w Pythonie na dużą skalę, należy koniecznie przejrzeć strony PyPI w witrynie <http://www.python.org>, czy po prostu internet. Praktyczne stosowanie Pythona to znacznie szerszy temat niż jego poznawanie, dlatego odsyłam do innych źródeł poświęconych temu zagadnieniu.

Podsumowanie rozdziału

Niniejszy rozdział zakończył część książki poświęconą wyjątkom omówieniem instrukcji związanych z tym zagadnieniem. Przyjrzelismy się często stosowanym przypadkom użycia wyjątków, a także zapoznaliśmy się z krótkim omówieniem często wykorzystywanych narzędzi programistycznych.

Rozdział ten kończy również główną część książki. W tym punkcie Czytelnikom przedstawiono pełny zbiór narzędzi Pythona wykorzystywany przez większość programistów. Tak naprawdę każda osoba, która dotarła aż tutaj, może się swobodnie uznawać za *oficjalnego programistę Pythona*. Przy najbliższej okazji należy rozważyć zakupienie koszulki z takim napisem (nie można też zapomnieć o umieszczeniu w CV informacji o znajomości języka Python).

Kolejna i zarazem ostatnia część książki to zbiór rozdziałów omawiających zagadnienia zaawansowane, które jednak mieszą się w kategoriach rdzenia języka. Rozdziały te są *lekturą opcjonalną*, ponieważ nie każdy programista Pythona musi zagłębiać się w przedstawione tam zagadnienia. Tak naprawdę większość osób może zatrzymać się już tutaj i zacząć poznawać role Pythona w rozmaitych dziedzinach zastosowania. Szczerze mówiąc, biblioteki aplikacji wydają się w praktyce bardziej istotnymi zagadnieniami niż zaawansowane (i dla niektórych osób egzotyczne) opcje języka.

Z drugiej strony, dla osób, które muszą zajmować się takimi kwestiami, jak dane binarne czy Unicode, mają do czynienia z narzędziami służącymi do budowania API, takimi jak deskryptory, dekoratory oraz metaklasy, albo po prostu chcą pogłębić swoją wiedzę, kolejna część książki będzie dobrym punktem wyjścia. Większe przykłady z tej części umożliwiają także zobaczenie zastosowania omówionych już koncepcji w nieco bardziej realistyczny sposób.

Ponieważ jest to koniec głównej części książki, nie będę przesadzał z trudnością quzu — tym razem zawiera on tylko jedno pytanie. Jak zawsze należy jednak pamiętać o wykonaniu ćwiczeń

końcowych w celu utrwalenia tego, czego nauczyliśmy się w ostatnich kilku rozdziałach. Ponieważ kolejna część jest lekturą opcjonalną, jest to ostatnia seria ćwiczeń kończących część książki. Osoby, które chcą zobaczyć przykłady połączenia tego, czego się nauczyliśmy, w prawdziwych skryptach pochodzących z często stosowanych aplikacji, odsyłam do rozwiązania ćwiczenia 4. w dodatku D.

Jeżeli na tym kończy się Twoja przygoda z tą książką, zatrzymaj się na podrozdziale „Bis” na końcu rozdziału 41., ostatniego w tej książce. (Z uwagi na czytelników, którzy przejdą do części „Zagadnienia zaawansowane”, nie mogę napisać, co się tam znajduje).

Sprawdź swoją wiedzę — quiz

1. (To pytanie jest powtórką z pierwszego quizu z rozdziału 1. — mówię, że będzie łatwo! :-)). Dlaczego mielonka pojawia się w tak wielu przykładach kodu Pythona w książce oraz internecie?

Sprawdź swoją wiedzę — odpowiedzi

1. Ponieważ Python został nazwany na cześć brytyjskiej grupy komediowej Monty Python (na podstawie przeprowadzonych w czasie moich szkoleń ankiet mogę powiedzieć, że w świecie Pythona jest to aż zbyt dobrze utrzymywana tajemnica!). Odniesienie do mielonki (ang. *spam*) pochodzi ze skeczu grupy Monty Python, w którym para próbująca zamówić coś do jedzenia w kafeterii zostaje zagłuszona przez chór Wikingów śpiewających piosenkę o mielonce. I gdybym mógł wstawić tutaj plik dźwiękowy z tą piosenką, z pewnością bym to zrobił...

Sprawdź swoją wiedzę — ćwiczenia do części siódmej

Ponieważ dotarliśmy do końca tej części książki, czas na kilka ćwiczeń związanych z wyjątkami, które pozwolą nam sprawdzić naszą znajomość podstaw. Wyjątki są naprawdę prostym narzędziem. Jeśli rozwiążemy poniższe ćwiczenia, oznacza to, że doskonale je opanowaliśmy. Rozwiązania można znaleźć w podrozdziale „Część VII Wyjątki oraz narzędzia” w dodatku D.

1. *Instrukcja try/except*. Należy napisać funkcję o nazwie `oops` w jawnym sposobie zgłaszającą po wywołaniu wyjątek `IndexError`. Później należy napisać kolejną funkcję wywołującą `oops` wewnętrznie instrukcji `try/except` w celu przechwycenia błędu. Co się stanie, jeśli zmienimy funkcję `oops` w taki sposób, by zgłaszała ona wyjątek `KeyError` zamiast `IndexError`? Skąd pochodzą nazwy `KeyError` oraz `IndexError`? Wskazówka: warto przypomnieć, że wszystkie nazwy bez składni kwalifikującej pochodzą z jednego z czterech zakresów.
2. *Obiekty wyjątków oraz listy*. Należy zmodyfikować napisaną wyżej funkcję `oops` w taki sposób, by zgłaszała zdefiniowany przez nas wyjątek o nazwie `MyError`. Wyjątek należy zidentyfikować za pomocą klasy (to obowiązek, chyba że jest wykorzystywana

wersja 2.5 lub starsza). Następnie należy rozszerzyć instrukcję `try` w funkcji przechwytyjącej w taki sposób, by oprócz `IndexError` przechwytywała ona ten wyjątek oraz jego instancję, a także wyświetlała przechwytywaną instancję.

3. *Obsługa błędów.* Należy napisać funkcję o nazwie `safe(func, *pargs, **kargs)`, która wykonuje dowolną funkcję z dowolną liczbą argumentów za pomocą składni wywołania z dowolną liczbą argumentów `*nazwa`), przechwytuje każdy wyjątek zgłoszony w czasie wykonywania funkcji i wyświetla ten wyjątek za pomocą wywołania `exc_info` z modułu `sys`. Następnie należy wykorzystać funkcję `safe` do wykonania funkcji `oops` z pierwszych dwóch ćwiczeń. Funkcję `safe` należy umieścić w pliku modułu o nazwie `exctools.py` i przekazać do niej interaktywnie funkcję `oops`. Jaki rodzaj komunikatu o błędzie otrzymamy? Wreszcie należy rozbudować funkcję `safe` w taki sposób, by kiedy wystąpi błąd, wyświetlała ona również ślad stosu Pythona, wywołując wbudowaną funkcję `print_exc` ze standardowego modułu `traceback` (szczegółowe informacje na jej temat można znaleźć w dokumentacji biblioteki Pythona). Funkcję `safe` można zakodować jako *dekorator*, wykorzystując techniki opisane w rozdziale 32., ale aby dowiedzieć się dokładnie, jak to się robi, musielibyśmy przejść do następnej części książki (rozwiążanie ćwiczenia da pogląd).
4. *Przykłady do samodzielnego przestudiowania.* Pod koniec dodatku D zamieściłem kilka przykładowych skryptów utworzonych jako ćwiczenia grupowe na kursach z Pythona, by każdy przestudiował je samodzielnie i wykonał w połączeniu ze zbiorzem dokumentacji Pythona. Nie są one opisane i wykorzystują narzędzia z biblioteki standardowej Pythona, z którymi trzeba będzie zapoznać się samodzielnie. Wielu osobom powinno to pomóc zobaczyć, w jaki sposób zagadnienia omówione w książce łączą się ze sobą w prawdziwych programach. Jeśli zaostrzy to nasz apetyt, ogromną liczbę większych i bardziej zaawansowanych programów w Pythonie na poziomie aplikacji można znaleźć w książkach będących kontynuacją tej, takich jak *Programming Python*, a także w internecie.

[1] Powiązane wywołanie `os._exit` również kończy program, jednak za pomocą natychmiastowego zakończenia — pomija działania czyszczące i nie może być przechwytywane za pomocą bloków `try/except` lub `try/finally`. Zazwyczaj jest wykorzystywane jedynie w procesach potomnych, zagadnieniu wykraczającym poza zakres niniejszej książki. Więcej szczegółów na ten temat można znaleźć w dokumentacji biblioteki lub bardziej zaawansowanych teksthach.

Część VIII Zagadnienia zaawansowane

Rozdział 37. Łańcuchy znaków Unicode oraz łańcuchy bajtowe

Temat ciągów znaków celowo nie został wyczerpany aż do tego miejsca. Typ ciągów i plików Unicode został krótko i bez wchodzenia w szczegóły przedstawiony w rozdziale 4. W rozdziale poświęconym łańcuchom znaków w części o typach podstawowych Pythona (rozdział 7.) celowo ograniczyłem omawiane zagadnienia do tych, które muszą znać wszyscy programiści Pythona. Ponieważ większość z nich pracuje z prostymi formami tekstu, takimi jak ASCII, w zupełności wystarcza im podstawowy typ `str` Pythona i powiązane z nim operacje, nie muszą więc zajmować się bardziej zaawansowanymi koncepcjami związanymi z łańcuchami znaków. Tak naprawdę ci programiści mogą w dużej mierze zignorować zmiany łańcuchów znaków w Pythonie 3.x i kontynuować wykorzystanie tego typu w taki sposób jak w przeszłości.

Z drugiej istnieją strony programiści zajmujący się bardziej wyspecjalizowanymi typami danych, jak znaki spoza zbioru ASCII czy zawartość plików graficznych. Na ich potrzeby (oraz potrzeby innych osób, które mogą kiedyś do nich dołączyć) w niniejszym rozdziale uzupełnimy informacje o łańcuchach znaków Pythona i przyjrzymy się pewnym bardziej zaawansowanym koncepcjom z nimi związanym.

W szczególności omówimy podstawy obsługi *tekstu Unicode* w Pythonie — łańcuchów znaków wykorzystywanych w aplikacjach zinternacjonalizowanych — a także *danych binarnych* — łańcuchów reprezentujących bezwzględne wartości bajtowe. Jak zobaczymy, zaawansowane zagadnienia związane z łańcuchami znaków w nowszych wersjach Pythona zostały zróżnicowane:

- *Python 3.x* udostępnia alternatywny typ łańcucha znaków dla danych binarnych i obsługuje tekst Unicode w normalnym typie łańcucha znaków (ASCII traktowane jest jako prosty rodzaj Unicode).
- *Python 2.x* udostępnia alternatywny typ łańcucha znaków dla tekstu Unicode poza ASCII, a w normalnym typie łańcucha znaków obsługuje zarówno prosty tekst, jak i dane binarne.

Dodatkowo, ponieważ model łańcuchów znaków Pythona ma bezpośredni wpływ na to, w jaki sposób przetwarzamy *pliki* spoza ASCII, omówimy tutaj podstawy również tego zagadnienia. Wreszcie przyjrzymy się krótko niektórym bardziej zaawansowanym *narzędziom* do obsługi łańcuchów znaków oraz danych binarnych, takim jak dopasowywanie do wzorców, serializacja obiektów, pakowanie danych binarnych, analiza składniowa XML, a także sposobom, w jakie na narzędzia te wpłynęły zmiany łańcuchów znaków z wersji 3.x.

Oficjalnie jest to rozdział poświęcony zagadnieniom zaawansowanym, ponieważ nie wszyscy programiści będą musieli zagłębić się w świat kodowania Unicode czy danych binarnych. Dla części czytelników wystarczy wprowadzenie z rozdziału 4., pozostały mogą odłożyć lekturę tego rozdziału na przyszłość. Jeśli jednak ktoś będzie chciał przetwarzać dane obu typów, szybko zobaczy, że model łańcuchów znaków Pythona obsługuje to, co trzeba.

Zmiany w łańcuchach znaków w Pythonie 3.x

Jedną z najbardziej zauważalnych zmian w wersji 3.x jest zmiana w typach obiektów łańcuchów znaków. W skrócie typy `str` i `unicode` z wersji 2.x złączyły się w typy `str` oraz `bytes` z wersji 3.x; dodany został nowy zmienny typ `bytearray`. Typ `bytearray` jest także dostępny w Pythonie 2.6 i 2.7 (choć nie we wcześniejszych wersjach), jednak został tam przeniesiony z wersji 3.x i nie rozróżnia w sposób tak wyraźny tekstu i zawartości binarnej.

Zmiany te mogą w znacznym stopniu wpływać na nasz kod, zwłaszcza jeśli przetwarzamy dane Unicode lub binarne. Mówiąc ogólnie, to, czy zagadnienie to ma dla nas znaczenie, w dużej mierze uzależnione jest od tego, do której z poniższych kategorii się zaliczamy:

- Jeśli zajmujemy się *tekstem Unicode* spoza zbioru ASCII — na przykład w kontekście zinternacjonalizowanych aplikacji czy wyniku niektórych analizatorów plików w formatach XML i JSON oraz baz danych — obsługa kodowania tekstu w 3.x zmieni się dla nas, jednak jednocześnie będzie też bardziej bezpośrednia, nieroóżnialna i dostępna w porównaniu z sytuacją z Pythona 2.x.
- Jeśli zajmujemy się *danymi binarnymi* — na przykład w postaci plików graficznych bądź audio czy spakowanych danych przetwarzanych za pomocą modułu `struct` — będziemy musieli opanować nowy obiekt `bytes` z Pythona 3.x, a także inne i ostrzejsze rozróżnienie pomiędzy danymi i plikami tekstowymi a binarnymi.
- Jeśli nie mieścimy się w *żadnej* z dwóch poprzednich kategorii, łańcuchów znaków w Pythonie 3.x będziemy w dużej mierze używać w taki sam sposób jak w wersji 2.x, z ogólnym typem łańcucha znaków `str`, plikami tekstowymi i wszystkimi znymi operacjami na tego typu obiektach, jakie już omówiliśmy. Wykorzystywane przez nas łańcuchy znaków będą kodowane i rozkodowywane za pomocą domyślnego rodzaju kodowania wykorzystywanej platformy (na przykład ASCII, UTF-8 lub Latin-1 — jeśli ktoś ma ochotę to sprawdzić, funkcja `sys.getpreferredencoding(False)` zwraca wartość domyślną); najprawdopodobniej nawet tego jednak nie zauważymy.

Innymi słowy, jeśli pisany przez nas tekst jest zawsze ASCII, wystarczą nam normalne obiekty łańcuchów znaków i plików tekstowych, a większość omawianych tu zagadnień możemy pominąć. Jak zobaczymy za chwilę, ASCII jest po prostu typem Unicode i podziorem innych rodzajów kodowania, dlatego jeśli nasze programy przetwarzają tekst ASCII, operacje na łańcuchach znaków i plikach po prostu będą działać.

Mimo to jednak, nawet jeśli mieścimy się w ostatniej z trzech przedstawionych wyżej kategorii, podstawowe zrozumienie modelu łańcuchów znaków Pythona z wersji 3.x może pomóc zarówno wytłumaczyć pewne zachowania tego typu, jak i ułatwić późniejsze opanowanie zagadnień związanych z Unicode czy danymi binarnymi w przyszłości.

Należy wyraźnie zaznaczyć, że czy tego chcemy, czy nie, przetwarzanie danych Unicode będzie w niedalekiej przyszłości nieodłączną częścią kodowania międzynarodowych aplikacji, z czym ostatecznie będzie musiał się pogodzić każdy programista. Tworzenie tego rodzaju aplikacji wykracza poza zakres tej książki, ale jeżeli ktoś ma do czynienia z internetem, plikami, katalogami, sieciami, bazami danych, potokami, formatami JSON i XML, a nawet graficznymi interfejsami, to obsługa formatu Unicode w Pythonie 3.x nie jest już dla niego tematem nieobowiązkowym.

Obsługa Unicode oraz danych binarnych z Pythona 3.x jest dostępna w wersji 2.x, jednak w innej postaci. Choć w niniejszym rozdziale skupimy się przede wszystkim na typach łańcuchów znaków z wersji 3.x, przy okazji przedstawimy także niektóre różnice z Pythonem 2.x. Bez względu na wykorzystywaną wersję tego języka omawiane tutaj narzędzia mogą mieć duże znaczenie w wielu typach programów.

Podstawy łańcuchów znaków

Zanim przyjrzymy się jakiemukolwiek kodowi, zacznijmy od ogólnego omówienia modelu łańcuchów znaków Pythona. By zrozumieć, dlaczego wersja 3.x została tak zmodyfikowana w tym zakresie, musimy zacząć od omówienia tego, w jaki sposób znaki są naprawdę reprezentowane w komputerze — zarówno zapisywane w plikach, jak i w pamięci.

Kodowanie znaków

Większość programistów myśli o łańcuchach znaków jako o seriach znaków wykorzystywanych do reprezentowania danych tekstowych. Sposób przechowywania znaków w pamięci komputera może się jednak różnić w zależności od zbioru znaków, jaki musi zostać zapisany. Na przykład format tekstu zapisanego w pliku zależy od użytego zestawu znaków.

Zestaw znaków jest to norma przypisująca poszczególnym znakom liczby całkowite, które można zapisywać w pamięci komputera. Na przykład w Stanach Zjednoczonych Ameryki został opracowany standard *ASCII*, który definiuje tekstowe łańcuchy znaków w rozumieniu większości amerykańskich programistów. ASCII definiuje znaki od 0 do 127 i pozwala, by każdy znak mógł być przechowywany w jednym 8-bitowym bajcie (z którego tak naprawdę wykorzystywanych jest jedynie siedem bitów).

Przykładowo standard ASCII odwzorowuje znak 'a' na wartość liczbową 97 (0x61 w notacji szesnastkowej), która zostaje przechowana w pojedynczym bajcie w pamięci i plikach. Jeśli chcemy zobaczyć, jak to działa, funkcja wbudowana Pythona `ord` zwraca wartość binarną znaku, natomiast funkcja `chr` zwraca znak dla podanej wartości kodu liczbowego:

```
>>> ord('a')                                # 'a' jest w ASCII bajtem z wartością  
binarną 97  
97  
>>> hex(97)  
'0x61'  
>>> chr(97)                                 # Wartość binarna 97 oznacza znak 'a'  
'a'
```

Czasami jednak jeden bajt na znak nie wystarczy. Różne symbole i litery ze znakami diakrytycznymi nie mieścią się w zakresie znaków zdefiniowanych przez ASCII. By uwzględnić znaki specjalne, niektóre standardy dopuszczają, by w 8-bitowym bajcie wszystkie możliwe wartości od 0 do 255 reprezentowały znaki, natomiast wartości od 128 do 255 (poza zakresem ASCII) były znakami specjalnymi.

Jeden z takich standardów, znany pod nazwą *Latin-1*, wykorzystywany jest powszechnie w Europie Zachodniej. W Latin-1 kody znaków powyżej 127 przypisane są do liter ze znakami diakrytycznymi i innych znaków specjalnych. Na przykład znak przypisany do wartości bajtowej 196 jest specjalną literą spoza zbioru ASCII:

```
>>> 0xC4  
196  
>>> chr(196)  
'Ä'
```

Standard ten pozwala na reprezentowanie szerokiej gamy dodatkowych znaków specjalnych. Mimo to niektóre alfabety definiują liczbę dodatkowych znaków uniemożliwiającą reprezentowanie ich wszystkich jako pojedynczych bajtów. Standard *Unicode* pozwala na większą elastyczność. W przypadku tekstu Unicode każdy znak może być reprezentowany za pomocą kilku bajtów. Unicode zazwyczaj wykorzystywany jest w programach

zinternacjonalizowanych i reprezentuje zbiory znaków europejskich oraz azjatyckich mające więcej znaków, niż da się reprezentować za pomocą bajtów 8-bitowych.

By przechować taki tekst w pamięci komputera, mówimy, że znaki tłumaczone są na „surowe” bajty i z powrotem za pomocą *kodowania* — reguł przekładania łańcucha znaków Unicode na sekwencję bajtów, a także ekstrakcji łańcucha znaków z sekwencji bajtów. Z punktu widzenia procedur takie tłumaczenie pomiędzy bajtami a łańcuchami znaków definiuje się za pomocą dwóch pojęć:

- *Kodowanie* to proces przekładania łańcucha znaków na jego odpowiednik w postaci surowych bajtów, zgodnie z pożądaną nazwą kodowania.
- *Dekodowanie (odkodowanie)* to proces przekładania łańcucha surowych bajtów na postać łańcucha znaków, zgodnie z pożądaną nazwą kodowania.

Oznacza to zatem, że *kodujemy* łańcuch znaków na surowe bajty i *dekodujemy* surowe bajty na łańcuchy znaków. Znaki w skryptach są zwykłymi bajtami zapisywanyimi w pamięci. Jeżeli jednak znaki te trzeba zapisać w pliku, przesyłać przez sieć, umieścić w dokumencie lub bazie danych, wtedy należy je zakodować zgodnie z jednym z wielu standardów.

W przypadku niektórych typów kodowania sam proces tłumaczenia jest trywialny — przykładowo ASCII i Latin-1 odwzorowują każdy znak na pojedynczy bajt, dzięki czemu żaden przekład nie jest konieczny. W przypadku innych typów kodowania odwzorowania mogą być bardziej skomplikowane i wymagać kilku bajtów na znak.

Szeroko wykorzystywane kodowanie *UTF-8* pozwala na przykład na reprezentowanie szerokiej gamy znaków, wykorzystując rozwiążanie ze *zmienną liczbą bajtów*. Kody znaków mniejsze od 128 reprezentowane są przez pojedynczy bajt; kody pomiędzy 128 a 0x7ff (2047) zamieniane są na dwa bajty, gdzie każdy bajt ma wartość pomiędzy 128 a 255. Kody większe od 0x7ff zamieniane są w sekwencje trzy- lub czterobajtowe o wartościach pomiędzy 128 a 255. Dzięki temu łańcuchy znaków ASCII są zwięzłe, omijane są problemy dotyczące kolejności bajtów, a także unikane bajty zerowe, które mogą spowodować problemy dla bibliotek języka C oraz pracy w sieci.

Ponieważ tabele odwzorowań znaków różnych typów kodowania z uwagi na zgodność przypisują znaki do tych samych kodów, ASCII jest *podzbiorem* zarówno Latin-1, jak i UTF-8. Oznacza to, że poprawny łańcuch znaków ASCII jest także poprawnym łańcuchem znaków w kodowaniu Latin-1 oraz UTF-8. Tak samo jest również, gdy dane przechowywane są w plikach — każdy plik ASCII jest poprawnym plikiem UTF-8, ponieważ ASCII jest 7-bitowym podzbiorem UTF-8.

I odwrotnie, kodowanie UTF-8 jest zgodne binarnie z ASCII dla wszystkich kodów znaków mniejszych od 128. Latin-1 i UTF-8 pozwalają po prostu na dodatkowe znaki — Latin-1 na znaki odwzorowane na wartości od 128 do 255 w ramach bajta, a UTF-8 na znaki, które mogą być reprezentowane za pomocą kilku bajtów.

Inne rodzaje kodowania pozwalają zapisywać bogatsze zbiory znaków w inny sposób. Na przykład w formatach *UTF-16* i *UTF-32* każdy znak, nawet taki, który zmieściłby się w jednym bajcie, jest reprezentowany odpowiednio przez dwa lub cztery bajty. W innych schematach stosowane są dodatkowo prefiksy oznaczające kolejność bajtów.

Aby zobaczyć, na czym to polega, można użyć metody *encode* zwracającej zadany ciąg znaków zakodowany w wybranym schemacie. Dwuznakowy ciąg zakodowany według schematu ASCII, Latin-1 i UTF-8 zajmuje 2 bajty. W przypadku schematu UTF-16 lub UTF-32 bajtów jest więcej, a dodatkowo stosowany jest nagłówek:

```
>>> S = 'ni'  
>>> S.encode('ascii'), S.encode('latin1'), S.encode('utf8')  
(b'ni', b'ni', b'ni')
```

```
>>> S.encode('utf16'), len(S.encode('utf16'))
(b'\xff\xfen\x00i\x00', 6)
>>> S.encode('utf32'), len(S.encode('utf32'))
(b'\xff\xfe\x00\x00n\x00\x00\x00i\x00\x00\x00\x00', 12)
```

W wersji Pythona 2.x uzyskane wyniki są inne (przed ciągami znaków nie jest umieszczana litera b). Wszystkie wymienione schematy — ASCII, Latin-1 i UTF-8 — a także wiele innych uznawane są za Unicode.

W przypadku programistów Pythona kodowanie określane jest jako łańcuch znaków zawierający jego nazwę. Python zawiera około stu różnych rodzajów kodowania — pełną listę można znaleźć w dokumentacji biblioteki tego języka. Zimportowanie modułu encodings i wykonanie help(encodings) pokazuje wiele z nazw typów kodowania. Niektóre z nich są zaimplementowane w języku Python, inne w C. Ponadto niektóre rodzaje kodowania mają kilka nazw. Przykładowo *latin-1*, *iso_8859_1* i *8859* są synonimami tego samego kodowania — Latin-1. Do kodowania powróćmy w dalszej części niniejszego rozdziału, przy omawianiu technik zapisywania łańcuchów znaków Unicode w skrypcie.

Więcej informacji na temat Unicode można znaleźć w dokumentacji biblioteki standardowej Pythona. W dziale „Python HOWTOs” znajduje się artykuł „Unicode HOWTO” zawierający dodatkowe informacje, które tutaj, z uwagi na brak miejsca, pominiemy.

Jak Python zapisuje ciągi znaków w pamięci

Kodowanie opisane w poprzednim podrozdziale dotyczy w rzeczywistości zapisywania tekstów w plikach i przesyłania ich na zewnątrz za pomocą różnych mediów. Jednak w pamięci tekst jest zawsze zapisywany w neutralnym formacie, w którym każdemu znakowi jest przypisywany jeden lub kilka znaków. Operacje na tekstu wykonywane są z uwzględnieniem takiego wewnętrznego, jednolitego formatu. Format jest zmieniany tylko podczas zapisywania i odczytywania tekstu z pliku, konwertowania go na ciągi bajtów lub przetwarzania za pomocą interfejsów API wymagających określonego kodowania. Jednak ciągi zapisywane w pamięci nie są kodowane. Są to po prostu obiekty opisane w tej książce. Z punktu widzenia kodu nie ma to znaczenia, jednak dla części czytelników takie ujęcie może być bardziej zrozumiałe.

Sposób zapisywania tekstu w pamięci ewoluował na przestrzeni czasu i w wersji 3.3 uległ istotnym zmianom:

Wersje 3.2 i starsze:

W tych wersjach tekst był kodowany w formacie UTF-16, w którym każdy znak zajmował 2 bajty (de facto był to standard UCS-2). Opcjonalnie można było stosować standard UCS-4, w którym każdy znak zajmuje 4 bajty pamięci.

Wersje 3.3 i nowsze:

W tych wersjach stosowany jest format o zmiennej długości, w którym każdy znak zajmuje 1, 2 lub 4 bajty w zależności od treści ciągu. Format jest dobierany odpowiednio do największego kodu Unicode spośród wszystkich znaków w ciągu. Dzięki takiemu schematowi oszczędza się w większości przypadków pamięć. Ponadto w każdym systemie operacyjnym można skonfigurować format UCS-4.

Stosowany w wersji 3.3 nowy schemat kodowania jest oszczędniejszy w porównaniu z wcześniejszymi stosowanymi odmianami standardu Unicode. Według dokumentacji tekst w zależności od treści zajmuje od dwóch do czterech razy mniej pamięci. Ponadto nie trzeba konwertować formatu ASCII na UTF-8, ponieważ znaki w obu formatach są reprezentowane w taki sam sposób. Operacje powielania znaków i wyodrębniania fragmentów ciągów z tekstu zapisanego w formacie ASCII są szybsze 4 razy, w formacie UTF-8 od dwóch do czterech razy, a w formacie

UTF-16 nawet 10 razy. Według niektórych pomiarów w wersji 3.3 ciągi zajmują od dwóch do trzech razy mniej pamięci niż w wersji 3.2, czyli mniej więcej tyle samo, co w wersji 2.7, mniej dostosowanej do standardu Unicode.

Jak wspomniałem w rozdziale 6., niezależnie od stosowanego schematu kodowania tekst zapisany w formacie Unicode należy traktować jako ciąg *znaków*, a nie *bajtów*. Jest to duża nowość dla programistów przyzwyczajonych od prostego formatu ASCII, w którym każdemu znakowi odpowiadał jeden bajt. Ta zasada już nie obowiązuje ani w odniesieniu do narzędzi przetwarzających teksty, ani do wielkości pamięci zajmowanej przez znak.

Narzędzia tekstowe

Obecnie zawartość ciągu i jego długość należy odnosić do formatu Unicode, w którym poszczególnym znakom przypisywane są liczby porządkowe. Na przykład wbudowana funkcja `ord` zwraca przypisany znakowi kod, który nie musi być kodem ASCII, choć może (ale nie musi) być liczbą 8-bitową. Podobnie funkcja `len` zwraca liczbę znaków, a nie bajtów w ciągu. Tekst zajmuje zazwyczaj więcej bajtów pamięci, niż wynosi liczba znaków, z których się składa.

Wielkość zajmowanego miejsca

Jak widzieliśmy w przykładzie w rozdziale 4., znak zapisany w formacie Unicode nie zawsze zajmuje jeden bajt, czy to w pliku, czy w pamięci. Nawet znaki ze zbioru ASCII, każdy kodowany za pomocą 7 bitów, nie muszą zajmować po jednym bajcie. W formacie UTF-16 znak zapisany w pliku zajmuje kilka bajtów, a w kodzie w Pythonie zajmuje 1, 2 lub 4 bajty pamięci. Jeżeli tekst traktuje się jako zbiór znaków, nie trzeba zajmować się szczególnymi zapisywania go w zewnętrznym lub wewnętrznym medium.

Kluczowe znaczenie ma fakt, że schemat kodowania dotyczy głównie zapisywania tekstu w pliku i przesyłania go przez sieć. W przypadku tekstu zapiswanego w pamięci nie obowiązuje pojęcie kodowania. Tekst jest po prostu jednolitą sekwencją kodów Unicode. Ciąg znaków w Pythonie jest obiektem, który jest tematem następnego podrozdziału.

Typy łańcuchów znaków Pythona

Przechodząc do konkretów, Python udostępnia typy danych łańcuchów znaków reprezentujące w naszych skryptach tekst. Typy łańcuchów znaków wykorzystywane w skryptach uzależnione są od wykorzystywanej wersji Pythona.

Python 2.x ma ogólny typ łańcucha znaków reprezentujący dane binarne oraz prosty tekst 8-bitowy, taki jak ASCII, wraz z odrębnym typem reprezentującym wielobajtowy tekst Unicode:

- `str` reprezentuje tekst 8-bitowy oraz dane binarne,
- `unicode` reprezentuje tekst Unicode.

Dwa typy łańcuchów znaków w Pythonie 2.x różnią się od siebie (`unicode` zezwala na dodatkową wielkość znaków i zawiera obsługę kodowania i dekodowania), jednak zbiory ich działań w dużej mierze pokrywają się. Typ łańcucha znaków `str` w wersji 2.x wykorzystywany jest w przypadku tekstu, który można reprezentować za pomocą bajtów 8-bitowych (w standardzie ASCII lub Latin-1), a także danych binarnych reprezentujących bezwzględne wartości bajtów. W Pythonie 3.x można natomiast znaleźć trzy typy obiektów łańcuchów znaków — jeden na potrzeby danych tekstowych i dwa przeznaczone dla danych binarnych:

- `str` reprezentuje tekst Unicode (w tym ASCII),
- `bytes` reprezentuje dane binarne (w tym tekst),
- `bytearray` jest zmienną odmianą typu `bytes`.

Jak wspomniano wcześniej, typ `bytearray` jest również dostępny w Pythonie 2.6 i 2.7, jednak został tam po prostu przeniesiony z wersji 3.x, ma mniej działań charakterystycznych dla swojej zawartości i generalnie uznawany jest za typ z Pythona 3.x.

Po co są stosowane różne typy ciągów?

Wszystkie typy łańcuchów znaków z wersji 3.x obsługują podobny zbiór operacji, jednak pełnią one odmienne role. Najważniejszą zmianą leżącą u podstaw zmian z wersji 3.x było połączenie typów normalnego łańcucha znaków i łańcucha znaków Unicode z Pythona 2.x w jeden typ obsługujący tekst zwykły i Unicode. Programiści chcieli odejść od dychotomii łańcuchów znaków z wersji 2.x i sprawić, by przetwarzanie Unicode stało się bardziej naturalne. Biorąc pod uwagę to, że ASCII i inne typy tekstu 8-bitowego tak naprawdę są prostym rodzajem Unicode, takie połączenie wydaje się logiczne i rozsądne.

By to uzyskać, typ `str` z Pythona 3.x zdefiniowany został jako *niezmienna sekwencja znaków* (niekoniecznie bajtów), która może być albo normalnym tekstem jak ASCII, z jednym bajtem na znak, albo bardziej rozbudowanym zbiorem tekstu takim jak UTF-8 w Unicode, który może zawierać znaki z większą liczbą bajtów. Łańcuchy znaków przetwarzane przez nasz skrypt za pomocą tego typu są kodowane zgodnie z wartością domyślną wykorzystywanej platformy. Nazwy kodowania można także podać w sposób jawnym, tak by przekładać obiekty `str` za pomocą różnych schematów — zarówno w pamięci, jak i przy przenoszeniu ich do plików i z nich.

Choć nowy typ `str` z Pythona 3.x spowodował zamierzone połączenie zwykłych łańcuchów znaków i łańcuchów znaków Unicode, wiele programów nadal musi przetwarzać surowe dane binarne, które nie są kodowane zgodnie z jakimkolwiek formatem tekstowym. Pliki graficzne czy audio, a także spakowane dane wykorzystywane w interfejsach z urządzeniami czy programami języka C, które możemy przetwarzać za pomocą modułu `struct` Pythona, mieszkają się w tej właśnie kategorii.

By móc obsługiwać przetwarzanie prawdziwych danych binarnych, wprowadzono zatem nowy typ danych o nazwie `bytes`. Jest to *niemutowalna sekwencja* dodatkowych liczb 8-bitowych, które można wyświetlać w postaci znaków ASCII, jeżeli jest to możliwe. Choć jest to typ obiektowy, umożliwia wykonywanie niemal wszystkich operacji oferowanych przez typ `str`. Dostępne są metody operujące na ciągach i sekwencjach, a nawet sprawdzające zgodność ze wzorcem. Nie ma jednak metod formatujących. W wersji 2.x ogólny typ `str` pełnił rolę danych binarnych, ponieważ łańcuchy znaków były po prostu sekwencjami bajtów (odrębny typ `unicode` zajmował się łańcuchami znaków wielobajtowych).

Obiekt `bytes` z Pythona 3.0 jest tak naprawdę sekwencją niewielkich liczb całkowitych, z których każda mieści się w przedziale od 0 do 255. Indeksowanie obiektu `bytes` zwraca obiekt `int`, wycinek zwraca inny obiekt `bytes`, natomiast wykonanie wbudowanej funkcji `list` zwraca listę liczb całkowitych, a nie znaków. Po przetworzeniu za pomocą operacji zakładających istnienie znaków w przypadku `bytes` zakłada się, że zawartość tych obiektów to bajty zakodowane w ASCII (metoda `isalpha` zakłada, że każdy bajt jest kodem znaku ASCII). Co więcej, obiekty `bytes` wyświetlane są dla wygody jako łańcuchy znaków, a nie liczby całkowite.

A skoro już przy tym jesteśmy, programiści Pythona dodali również w wersji 3.0 typ `bytearray`. Typ `bytearray` jest odmienną `bytes`, która jest *zmienna*, dzięki czemu obsługuje modyfikację w miejscu. Obsługuje on zwykłe operacje na łańcuchach znaków, te same co typy `str` i `bytes`, a także wiele z operacji modyfikacji w miejscu obsługiwanych przez listy (na przykład metody `append` i `extend`, przypisywanie do indeksów). Jest to cenna cecha zarówno w przypadku danych binarnych, jak i prostych tekstowych. Zakładając, że łańcuchy znaków mogą być traktowane jako surowe bajty (np. kody ASCII lub Latin-1), `bytearray` wreszcie dodaje możliwość dokonywania bezpośrednich zmian w miejscu w przypadku danych będących łańcuchami znaków — coś, co było niemożliwe do uzyskania w Pythonie 2.x bez konwersji na zmienny typ danych i nie jest obsługiwane przez typy `str` oraz `bytes` z Pythona 3.x.

Choć wersje 2.x i 3.x oferują w dużej mierze tę samą funkcjonalność, w inny sposób ją łączą. Tak naprawdę odwzorowanie z typów łańcuchów znaków Pythona 2.x na typy z Pythona 3.x nie

jest bezpośrednie — `str` z 2.x równy jest `str` i `bytes` z 3.x, natomiast `str` z 3.x równy jest `str` i `unicode` z 2.x. Co więcej, unikalna jest zmienność typu `bytarray` z wersji 3.x.

W praktyce ta asymetria nie jest aż tak skomplikowana, jak mogłoby się wydawać. Sprowadza się do następującej kwestii: w wersji 2.x użyjemy `str` w przypadku prostego tekstu oraz danych binarnych, a `unicode` w przypadku bardziej zaawansowanych form tekstu. W Pythonie 3.x użyjemy `str` dla dowolnego rodzaju tekstu (prostego i Unicode), a `bytes` lub `bytarray` dla danych binarnych. W praktyce wybór jest często determinowany przez wykorzystywane przez nas narzędzia — zwłaszcza w przypadku narzędzi do przetwarzania plików, czyli tematu kolejnego podrozdziału.

Pliki binarne i tekstowe

Wejście-wyjście plików zostało w Pythonie 3.x przebudowane, by odzwierciedlać rozróżnienie między typami `str` i `bytes` oraz automatycznie obsługiwać kodowanie tekstu Unicode. Python tworzy teraz jasne, niezależne od platformy rozróżnienie między plikami tekstowymi a binarnymi:

Pliki tekstowe

Kiedy plik otwierany jest w *trybie tekstowym*, wczytanie jego danych automatycznie dekoduje jego zawartość (zgodnie z ustawieniem domyślnym platformy lub podaną nazwą kodowania) i zwraca ją w postaci obiektu `str`. Zapisanie do pliku odbywa się poprzez obiekt `str` i automatyczne kodowanie przed przeniesieniem do pliku. Podczas zapisywania i odczytywania plików stosowane jest kodowanie wskazane w argumencie lub domyślne, ustawione w systemie operacyjnym. Pliki w trybie tekstowym obsługują również uniwersalne przekładanie końca wiersza oraz dodatkowe argumenty określające kodowanie. W zależności od nazwy kodowania pliki tekstowe mogą również automatycznie przetwarzać sekwencję znacznika kolejności bajtów (ang. *byte order mark, BOM*) na początku pliku (więcej na ten temat za moment).

Pliki binarne

Kiedy plik otwierany jest w *trybie binarnym* za pomocą dodania `b` (małej litery) do argumentu łańcucha znaków trybu we wbudowanym wywołaniu `open`, wczytanie jego danych nie powoduje ich zdekodowania, a po prostu zwraca zawartość pliku w surowej i niezmienionej postaci, jako obiekt `bytes`. Zapisywanie do pliku odbywa się poprzez obiekt `bytes` i w podobny sposób zawartość przenoszona jest do pliku bez zmian. Pliki w trybie bajtowym przyjmują również obiekt `bytarray` w przypadku zawartości, która ma być do nich zapisana.

Ponieważ Python rozróżnia typy `str` i `bytes`, musimy zdecydować, czy nasze dane są z natury tekstowe, czy binarne, i użyć odpowiedniego obiektu w celu ich poprawnej reprezentacji w skrypcie. Wybór trybu otwarcia pliku decyduje o tym, jaki typ obiektu skrypt wykorzysta do reprezentowania swojej zawartości:

- Jeśli przetwarzamy pliki graficzne, spakowane dane utworzone za pomocą innych programów, których zawartość musimy poddać ekstrakcji, podobnie jak strumienie danych różnych urządzeń, istnieje spora szansa, że chcemy je potraktować, używając typu `bytes` oraz plików w *trybie binarnym*. Jeśli chcemy uaktualniać dane bez tworzenia ich kopii w pamięci, możemy także wybrać typ `bytarray`.
- Jeśli zamiast tego przetwarzamy coś, co z natury jest tekstowe, jak na przykład dane wyjściowe programu, kod HTML, tekst ze znakami międzynarodowymi, pliki CSV czy XML, najprawdopodobniej będziemy chcieli skorzystać z typu `str` i plików w *trybie tekstowym*.

Warto zauważyć, że argument *łańcucha znaków trybu* wbudowanej funkcji `open` (jej drugi argument) w Pythonie 3.x stał się bardzo istotny — jego wartość nie tylko określa *tryb*

przetwarzania pliku, ale także sugeruje typ obiektu Pythona. Dodając do łańcucha trybu b, wybieramy tryb binarny i otrzymamy (a także musimy dostarczyć) obiekt bytes, który będzie reprezentował zawartość pliku w czasie odczytywania i zapisywania. Bez znaku b plik przetwarzany jest w trybie tekstowym, a do reprezentowania jego zawartości w skrypcie posłuży nam obiekt str. Przykładowo tryby rb, wb oraz rb+ wymuszają obiekt bytes, natomiast r, w+ oraz rt (tryb domyślny) — obiekt str.

Pliki w trybie tekstowym mogą także obsługiwać sekwencję znacznika kolejności bajtów (BOM), która może pojawiać się na początku pliku w niektórych schematach kodowania. W kodowaniu UTF-16 oraz UTF-32 sekwencja BOM określa format *big-endian* lub *little-endian* (określający w przybliżeniu, który koniec łańcucha bitowego jest ważniejszy). Plik tekstowy UTF-8 może także zawierać sekwencję BOM w celu zadeklarowania, że jest zakodowany jako UTF-8, jednak nie ma gwarancji, że tak będzie. Przy zapisywaniu i odczytywaniu danych za pomocą tych schematów kodowania Python automatycznie pomija lub zapisuje BOM zgodnie z zasadami opisanymi w dalszej części rozdziału.

W Pythonie 2.x powyższe operacje wykonywane są w taki sam sposób, jednak metoda open jest wykorzystywana do otwierania i przetwarzania plików binarnych, a codecs.open do plików tekstowych zapisanych w formacie Unicode. Ta ostatnia również koduje i dekoduje znaki, o czym się przekonamy w dalszej części rozdziału. Najpierw jednak sprawdźmy dokładnie, jak funkcjonuje model danych Unicode w Pythonie.

Podstawy kodowania ciągów znaków

Przyjrzyjmy się kilku przykładom demonstrującym wykorzystywanie nowego typu łańcuchów znaków z Pythona 3.x. Jedna uwaga: kod w niniejszym podrozdziale został wykonany i ma zastosowanie jedynie do wersji 3.x. Mimo to podstawowe działania na łańcuchach znaków są zasadniczo przenośne pomiędzy wersjami Pythona. Proste łańcuchy znaków ASCII reprezentowane za pomocą typu str działają tak samo w wersji 2.x oraz 3.x (i dokładnie tak samo, jak widzieliśmy to w rozdziale 7. niniejszej książki). Co więcej, choć w Pythonie 2.x nie istnieje typ bytes (jest tam tylko ogólny typ str), zazwyczaj można wykonać w nim kod, który zakłada, że typ ten tam się znajduje. W wersjach 2.6 i 2.7 wywołanie bytes(X) obecne jest jako synonim dla str(X), a nowa forma literalu b'...' traktowana jest tak samo jak normalny literał łańcucha znaków '...'. W niektórych wyizolowanych przypadkach nadal możemy natrafić na pewne różnice w zakresie wersji — wywołanie bytes z Pythona 2.6/2.7 nie pozwala na przykład na podanie drugiego argumentu (nazwy kodowania) wymaganego przez bytes z wersji 3.x.

Literały tekstowe w Pythonie 3.x

Obiekty łańcuchów znaków Pythona 3.x pojawiają się, kiedy wywołujemy funkcję wbudowaną, taką jak str lub bytes, przetwarzamy plik utworzony za pomocą wywołania open (więcej informacji w kolejnym podrozdziale) lub zapisujemy w kodzie skryptu składnię literalu. W tym ostatnim przypadku nowa postać literalu b'xxx' (i jej równoważnik B'xxx') wykorzystywana jest do tworzenia obiektów typu bytes w Pythonie 3.x. Obiekty typu bytearray tworzą się, wywołując funkcję bytearray z różnymi argumentami.

Z formalnego punktu widzenia w Pythonie 3.x wszystkie obecne formy literałów — 'xxx', "xxx" oraz bloki w potrójnych cudzysłowach lub apostrofach — generują obiekt str. Dodanie znaku b lub B przed dowolną z nich tworzy zamiast tego obiekty bytes. Nowy literał b'...' jest podobny w formie do surowego łańcucha znaków r'...' wykorzystywanego do powstrzymania znaków ucieczki z ukośnikami lewymi. Rozważmy poniższy, wykonany w Pythonie 3.x, kod:

```
C:\code> c:\python30\python
```

```

>>> B = b'mielonka'          # W wersji 3.x utworzenie obiektu typu bytes (8-
bitowe bajty)

>>> S = 'jajka'            # W wersji 3.x utworzenie obiektu typu str (znaki
Unicode)

>>> type(B), type(S)
(<class 'bytes'>, <class 'str'>)

>>> B                      # Wyświetla jako łańcuchy znaków, a tak naprawdę są
to sekwencje liczb całkowitych

b'mielonka'

>>> S

'jajka'

```

Obiekt `bytes` jest tak naprawdę sekwencją krótkich liczb całkowitych, choć swoją zawartość wyświetla jako znaki zawsze, gdy jest to możliwe:

```

>>> B[0], S[0]              # Indeksowanie zwraca typ int dla bytes, typ str
dla str

(109, 'j')

>>> B[1:], S[1:]           # Wycinek tworzy inny obiekt bytes lub str

(b'ielonka', 'ajka')

>>> list(B), list(S)       # bytes to tak naprawdę liczby całkowite

([109, 105, 101, 108, 111, 110, 107, 97], ['j', 'a', 'j', 'k', 'a'])

```

Obiekt `bytes` jest niezmienny, podobnie jak `str` (choć opisany niżej `bytearray` już nie jest). Nie można przypisać obiektu `str`, `bytes` czy liczby całkowitej do wartości przesunięcia dla obiektu `bytes`.

```

>>> B[0] = 'x'             # Oba są niezmienne
TypeError: 'bytes' object does not support item assignment

>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment

```

Należy zwrócić uwagę, że przedrostki `b` i `B` można stosować z dowolną odmianą literału tekstowego, również wykorzystującą potrójne cudzysłowy. Uzyskuje się wtedy ciąg surowych bajtów, które niekoniecznie reprezentują znaki:

```

>>> B = B""""          # Przedrostek bytes działa dla apostrofów, cudzysłowów i
podwójnych apostrofów oraz cudzysłowów

... xxxx
... yyyy
... """"

>>> B
b'\nxxxx\nyyyy\n'

```

Literały Unicode w Pythonie 2.x i 3.3

Składnia `u'xxx'` i `U'xxx'`, stosowana w wersji Pythona 2.x do definiowania literałów Unicode, została uznana za niepotrzebną i usunięta w wersji 3.0. W wersjach 3.x standardem tekstowym jest Unicode. Aby jednak zapewnić kompatybilność ze starszymi wersjami języka, składnia ta z powrotem została wprowadzona w wersji 3.3 i służy do definiowania zwykłych ciągów typu `str`:

```
C:\code> C:\python33\python
>>> U = u'mielonka'                      # Literał Unicode z wersji 2.x akceptowany
w wersjach 3.3 i nowszych
>>> type(U)                             # Jest to zwykły typ str, ale kompatybilny
wstecz
<class 'str'>
>>> U
'spam'
>>> U[0]
's'
>>> list(U)
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a']
```

Powyższa składnia nie jest dostępna w wersjach od 3.0 do 3.2, w których stosuje się zapis `'xxx'`. Tę formę należy też stosować w kodzie pisany od razu w wersji 3.x, ponieważ składnia z wersji 2.x przestała obowiązywać. Składnię typową dla wersji 2.x można stosować w przypadku przenoszenia kodu ze starszej wersji do 3.3 lub nowszej, ponieważ dzięki temu zadanie to jest łatwiejsze, jak również zapewnia się w ten sposób kompatybilność kodu (patrz przykład z walutami z rozdziału 25.). Tak czy owak w wersji 3.x tekst jest zawsze kodowany w standardzie Unicode, nawet jeżeli składa się wyłącznie ze znaków ASCII (więcej informacji na temat zapisywania tekstu Unicode spoza ASCII znajduje się w podrozdziale „Kod tekstu spoza zakresu ASCII”).

Literały tekstowe w Pythonie 2.x

Wszystkie trzy opisane w poprzednim podrozdziale formy kodowania ciągów znaków w Pythonie 3.x można stosować również w wersji 2.x, ale uzyskiwane efekty są wtedy inne. Jak wspomniałem wcześniej, w wersjach 2.6 i 2.7 literał bajtowy `b'xxx'` został wprowadzony w celu zapewnienia kompatybilności w przód z wersją 3.x. Oznacza on to samo co `'xxx'`, czyli ciąg typu `str` (prefiks `b` można pominąć), a słowo kluczowe `bytes` jest synonimem `str`. Natomiast w wersji 3.x, jak już widzieliśmy, każda z form definiuje inny ciąg znaków:

```
C:\code> C:\python27\python
>>> B = b'mielonka'                      # Literał bajtowy w wersji 3.x jest typem
str w wersjach 2.6/2.7
>>> S = 'jajka'                           # Typ str jest sekwencją bajtów (znaków)
>>> type(B), type(S)
(<type 'str'>, <type 'str'>)
>>> B, S
('mielonka', 'jajka')
>>> B[0], S[0]
```

```
('m', 'j')
>>> list(B), list(S)
([['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a'], ['j', 'a', 'j', 'k', 'a']])
```

W wersji 2.x specjalny literał Unicode służy do definiowania bogatszych ciągów znaków:

```
>>> U = u'mielonka'      # W wersji 2.x literał Unicode definiuje ciąg znaków
osobnego typu
>>> type(U)              # W wersji 3.x też można go stosować, ale uzyskuje się
zwykły ciąg typu str
<type 'unicode'>
>>> U
u'mielonka'
>>> U[0]
u'm'
>>> list(U)
[u'm', u'i', u'e', u'l', 'o', u'n', u'k', u'a']
```

Jak już wiemy, ze względu na konieczność zapewnienia kompatybilności ze starszymi wersjami powyższy zapis stosuje się również w wersji 3.3, ale skutkuje on uzyskaniem zwykłego ciągu typu `str` (prefiks `u` jest pomijany).

Konwersje typów ciągów

Choć Python 2.x pozwalał na swobodne mieszanie obiektów typu `str` i `unicode` (jeśli łańcuchy znaków zawierały jedynie 7-bitowy tekst ASCII), wersja 3.x zachowuje o wiele bardziej ścisły podział. Obiekty typów `str` i `bytes` nigdy automatycznie nie mieszą się ze sobą w wyrażeniach i nigdy nie są wzajemnie konwertowane przy przekazywaniu do funkcji. Funkcja oczekująca, że argument będzie obiektem `str`, nie przyjmie obiektu `bytes` — i odwrotnie.

Z tego powodu Python 3.x wymaga zdecydowania się na jeden z typów lub wykonania ręcznego i jawnego przekształcenia:

- Funkcje `str.encode()` oraz `bytes(S, kodowanie)` przekładają łańcuch znaków na jego postać bajtową i z obiektu `str` tworzą w rezultacie obiekt `bytes`.
- Funkcje `bytes.decode()` oraz `str(B, kodowanie)` przekładają łańcuch znaków na jego postać bajtową i z obiektu `bytes` tworzą w rezultacie obiekt `str`.

Powyższe metody `encode` i `decode` (a także opisana w następnym podrozdziale funkcja `open`) wykorzystują albo domyślne kodowanie platformy, albo nazwę kodowania przekazaną w sposób jawnego. Na przykład w Pythonie 3.x powyższe metody domyślnie wykorzystują kodowanie UTF-8, natomiast funkcja `open` wykorzystuje wartość z modułu `locale`, która może być różna w zależności od używanego systemu operacyjnego. W wersji 2.x wszystkie powyższe metody domyślnie wykorzystują standard ASCII, właściwy modułowi `sys` (można go zmienić na początku programu). Na przykład w wersji 3.x mamy:

```
>>> S = 'jajka'
>>> S.encode()                      # Ze str na bytes: kodowanie tekstu na
bajty
b'jajka'
```

```

>>> bytes(S, encoding='ascii')      # Ze str na bytes, alternatywa
b'jajka'
>>> B = b'mielonka'
>>> B.decode()                  # Z bytes do str: zdekodowanie bajtów na
tekst
'mielonka'
>>> str(B, encoding='ascii')      # Z bytes do str, alternatywa
'mielonka'

```

Dwie uwagi. Po pierwsze, kodowanie domyślne platformy dostępne jest w module sys, jednak argument kodowania dla bytes nie jest opcjonalny, nawet jeśli jest tak w przypadku str.encode (oraz bytes.decode).

Po drugie, choć wywołania str nie wymagają podania argumentu kodowania, tak jak to jest w przypadku bytes, opuszczenie go w wywołaniach str nie oznacza wartości domyślnej — wywołanie str bez kodowania zwraca *łańcuch wyświetlania* obiektu bajtowego, a nie postać po konwersji do str (i nie jest to zazwyczaj efekt, którego oczekujemy!). Zakładając, że zmienne B oraz S są tym samym co w poprzednim kodzie, mamy:

```

>>> import sys, locale          # Metoda open w systemie Windows
wykorzystuje stronę kodową cp1252
                                         # (nadzbiór Latin-1)

>>> sys.platform              # Klasa str nigdy nie używa ustawień
domyślnych
'win32'

>>> locale.getpreferredencoding(False), sys.getdefaultencoding()
('cp1252', 'utf-8')

>>> bytes(S)
TypeError: string argument without an encoding

>>> str(B)                    # Obiekt str bez kodowania
                                # Łąńcuch znaków wyświetlania, a nie
konwersja!

>>> len(str(B))
11

>>> len(str(B, encoding='ascii')) # Użycie kodowania w celu konwersji na str
8

```

Aby uniknąć wątpliwości, należy w wersji 3.x w argumencie funkcji podawać nazwę standardu kodowania, nawet jeśli jest to standard domyślny. Konwersje typów odbywają się tak samo jak w wersji 2.x, choć dzięki możliwości mieszania typów tekstowych nie jest konieczna konwersja tekstu zapisanego w standardzie ASCII. Poza tym nazwy narzędzi są różne w zależności od stosowanego typu tekstopowego. W wersji 2.x konwersja odbywa się pomiędzy niekodowanym typem str a kodowanym unicode, a w wersji 3.x pomiędzy niekodowanym typem bytes a kodowanym str:

```

>>> S, U = 'mielonka', u'jajka'          # Konwersja typów tekstowych w wersji
2.x

>>> S, U
('mielonka', u'jajka')

>>> unicode(S), str(U)                  # Konwersja str->unicode i unicode-
>str w wersji 2.x
(u'mielonka', 'jajka')

>>> S.decode(), U.encode()            # Konwersja bytes->str i str->bytes w
wersji 3.x
(u'mielonka', 'jajka')

```

Kod łańcuchów znaków Unicode

Kodowanie i dekodowanie mają większe znaczenie, kiedy zaczniemy zajmować się prawdziwym tekstem Unicode spoza zakresu ASCII. By umieścić w kodzie łańcucha znaków znaki Unicode, z których części pewnie nawet nie możemy wpisać na posiadanej klawiaturze, literały łańcuchów znaków Pythona obsługują zarówno szesnastkowe wartości bajtowe z sekwencjami ucieczki "`\xNN`", jak i sekwencje ucieczki Unicode w literałach łańcuchów znaków "`\uNNNN`" oraz "`\UNNNNNNNN`". W przypadku sekwencji ucieczki Unicode pierwsza postać podaje cztery cyfry szesnastkowe w celu zakodowania 2-bajtowego (16-bitowego) kodu znaku, natomiast druga forma podaje osiem cyfr szesnastkowych dla kodu 4-bajtowego (32-bitowego).

Kod tekstu z zakresu ASCII

Przyjrzyjmy się kilku przykładom demonstrującym podstawy kodowania tekstu. Jak widzieliśmy, tekst ASCII jest prostym typem Unicode, przechowanym jako sekwencja wartości bajtowych reprezentujących znaki:

```

C:\code> c:\python33\python

>>> ord('X')                         # 'X' ma wartość binarną 88 w kodowaniu
domyślnym

88

>>> chr(88)                          # 88 oznacza znak 'X'

'X'

>>> S = 'XYZ'                         # łańcuch znaków Unicode dla tekstu ASCII

>>> S

'XYZ'

>>> len(S)                            # Długość 3 znaków

3

>>> [ord(c) for c in S]              # 3 bajty z liczbowymi wartościami
porządkowymi

[88, 89, 90]

```

Taki normalny 7-bitowy tekst ASCII reprezentowany jest za pomocą jednego znaku na bajt w każdym ze schematów Unicode opisanych wcześniej w rozdziale:

```
>>> S.encode('ascii')          # Wartości 0..127, każde w 1 bajcie (7 bitach)
b'XYZ'

>>> S.encode('latin-1')       # Wartości 0..255, każde w 1 bajcie (8 bitach)
b'XYZ'

>>> S.encode('utf-8')         # Wartości 0..127, każde w 1 bajcie,
128..2047 w 2, inne w 3 lub 4
b'XYZ'
```

Tak naprawdę obiekty zwracane dzięki zakodowaniu tekstu ASCII w ten sposób są sekwencjami krótkich liczb całkowitych, które — tak się składa — gdy jest to możliwe, wyświetlane są jako znaki ASCII:

```
>>> S.encode('latin-1')
b'XYZ'

>>> S.encode('latin-1')[0]
88

>>> list(S.encode('latin-1'))
[88, 89, 90]
```

Kod tekstu spoza zakresu ASCII

Znaki inne niż ASCII można wpisywać na dwa sposoby:

- W wersji 3.x lub literałach Unicode w wersji 2.x (oraz 3.3 dla zapewnienia kompatybilności) stosować znak ucieczki *szesnastkowy* lub *Unicode* w podstawowych *literałach tekstowych*.
- Stosować *szesnastkowy* znak ucieczki w *bajtowych ciągach* znaków — podstawowych w wersji 2.x oraz bajtowych w wersji 3.x (oraz 2.x dla zapewnienia kompatybilności).

Należy pamiętać, że w ten sposób w tekstowym ciągu umieszcza się znaki o podanych kodach, a w ciągach bajtowych znaki w niekodowanej formie. Liczba reprezentująca znak w ciągu bajtowym jest taka sama jak kod Unicode w ciągu tekstowym tylko w przypadku niektórych znaków i rodzajów kodowania. Za pomocą szesnastkowych znaków ucieczki można kodować pojedyncze wartości bajtowe, natomiast znaki ucieczki Unicode umożliwiają wpisywanie znaków zajmujących dwa lub cztery bajty. Do utworzenia pojedynczego znaku spoza zestawu ASCII można użyć funkcji `chr`, podając w jej argumencie odpowiedni kod. Ponadto, jak się przekonamy później, na znaki stosowane w skrypcie ma również wpływ deklaracja umieszczana w kodzie źródłowym.

Wartości szesnastkowe `0xC4` i `0xE8` są na przykład kodami dwóch specjalnych znaków akcentowanych spoza 7-bitowego zakresu ASCII, jednak możemy je osadzać w obiektach `str` Pythona 3.x, ponieważ obsługuje on obecnie standard Unicode:

```
>>> chr(0xc4)          # 0xC4, 0xE8: znaki spoza zakresu ASCII
'Ä'
```

```

>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'                                # Jednobajtowe 8-bitowe szesnastkowe
sekwencje ucieczki

>>> S
'Äè'

>>> S = '\u00c4\u00e8'                            # 16-bitowe sekwencje ucieczki Unicode

>>> S
'Äè'

>>> len(S)                                     # 2 znaki długości (nie liczba bajtów!)

2

```

Należy zwrócić uwagę, że w literałach takich jak powyższe znaki ucieczki szesnastkowe i Unicode zawierają kody znaków, a nie wartości bajtowe. Po znaku x zawsze umieszcza się dwie cyfry oznaczające 8-bitowy kod znaku, natomiast po znaku u lub U odpowiednio cztery lub osiem cyfr oznaczających kody 16-bitowe i 32-bitowe:

```

>>> S = '\u000000c4\u000000e8'      # Dwa 8-cyfrowe (32-bitowe) kody znaków

>>> S
'Äè'

```

Jak widzieliśmy wcześniej, w Pythonie 2.x kodowanie wygląda podobnie, jednak znaki ucieczki Unicode można stosować tylko w literałach tego właśnie typu. W wersji 3.x są to zwykłe literały tekstowe, ponieważ są one zawsze kodowane w standardzie Unicode.

Kodowanie i dekodowanie tekstu spoza zakresu ASCII

Jeśli teraz spróbujemy zakodować łańcuch znaków spoza ASCII na łańcuch bajtowy wykorzystującą ASCII, otrzymamy błąd:

```

>>> S = '\u00c4\u00e8'      # Ciąg złożony z dwóch znaków spoza zakresu ASCII

>>> S
'Äè'

>>> len(S)

2

>>> S.encode('ascii')

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)

```

Można jednak wpisywać znaki w standardzie *Latin-1*, ponieważ każdy z nich ma 8-bitowy kod i w efekcie w ciągu zajmuje jeden bajt. Można również stosować standard *UTF-8*, ponieważ każdy znak spoza zbioru ASCII jest zapisywany w postaci dwubajtowych kodów Unicode. Takie ciągi zapisane w pliku składają się z bajtów zwróconych przez metodę `encode`, odzwierciedlających zadany sposób kodowania. Ilustruje to poniższy przykład:

```

>>> S.encode('latin-1')          # Jeden bajt na znak

```

```

b'\xc4\xe8'
>>> S.encode('utf-8')                      # Dwa bajty na znak
b'\xc3\x84\xc3\xa8'
>>> len(S.encode('latin-1'))                # 2 bajty w latin-1, 4 w utf-8
2
>>> len(S.encode('utf-8'))
4

```

Warto zauważyć, że możemy także postąpić odwrotnie, wczytując surowe bajty z pliku i dekodując je z powrotem na łańcuch znaków Unicode. Jak jednak zobaczymy później, tryb kodowania podawany do wywołań `open` powoduje automatyczne dekodowanie danych wejściowych (i uniknięcie problemów, które mogą się pojawić w związku z wczytaniem części sekwencji znaków przy wczytywaniu bloków bajtów):

```

>>> B = b'\xc4\xe8'                      # Tekst zakodowany w standardzie Latin-1.
>>> B
b'\xc4\xe8'
>>> len(B)                                # 2 surowe bajty, 2 znaki
2
>>> B.decode('latin-1')                    # Zdekodowanie do tekstu latin-1
'Àè'
>>> B = b'\xc3\x84\xc3\xa8'              # Tekst zakodowany w standardzie UTF-8
>>> len(B)                                # 4 surowe bajty
4
>>> B.decode('utf-8')                     # Zdekodowany tekst w standardzie UTF-8
'Àè'
>>> len(B.decode('utf-8'))                # 2 znaki Unicode
2

```

Inne techniki kodowania łańcuchów Unicode

Niektóre schematy kodowania wykorzystują jeszcze większe sekwencje bajtów do reprezentowania znaków. Jeśli jest to potrzebne, możemy podać 16- i 32-bitowe wartości Unicode dla znaków w łańcuchu — należy użyć zapisu "`\u....`" z czterema cyframi dla tego pierwszego i "`\U....`" z ośmioma cyframi szesnastkowymi dla tego drugiego:

```

>>> S = 'A\u00c4B\u000000e8C'
>>> S                                    # A, B, C i dwa znaki spoza ASCII
'AÄBèC'
>>> len(S)                              # 5 znaków długości
5

```

```
>>> S.encode('latin-1')
b'A\xc4B\xe8C'
>>> len(S.encode('latin-1'))    # 5 bajtów w Latin-1
5
>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> len(S.encode('utf-8'))      # 7 bajtów w UTF-8
7
```

Ciągi Unicode można tworzyć nie tylko za pomocą szesnastkowych znaków ucieczki, ale też funkcji chr. Jednak w przypadku dłuższych ciągów jest to żmudny sposób:

```
>>> S = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'
>>> S
'AÄBèC'
```

Co ciekawe, inne typy kodowania mogą wykorzystywać bardzo odmienne formaty bajtowe. Przykładowo kodowanie cp500 EBCDIC nie koduje nawet ASCII w ten sam sposób jak pozostałe (ponieważ Python koduje i dekoduje za nas, musimy się tym przejmować jedynie wtedy, gdy podajemy nazwy kodowania):

```
>>> S
'AÄBèC'

>>> S.encode('cp500')          # Dwa inne kodowania
zachodnioeuropejskie
b'\xc1c\xc2T\xc3'

>>> S.encode('cp850')          # 5 bajtów każdy
b'A\x8eB\x8aC'

>>> S = 'mielonka'            # Tekst ASCII jest taki sam w większości
typów kodowania
>>> S.encode('latin-1')
b'mielonka'
>>> S.encode('utf-8')
b'mielonka'
>>> S.encode('cp500')          # Ale nie w cp500: IBM EBCDIC!
b'\x94\x89\x85\x93\x96\x95\x92\x81'
>>> S.encode('cp850')
b'mielonka'
```

Ta sama zasada dotyczy kodowania UTF-16 i UTF-32, w których każdy znak zawsze zajmuje odpowiednio 2 lub 4 bajty. Znaki spoza zakresu ASCII kodowane są inaczej, a znaki mieszące się w tym zakresie nie są kodowane za pomocą jednego bajta:

```
>>> S = 'A\x00c4B\x000000e8C'
```

```

>>> S.encode('utf-16')
b'\xff\xfeA\x00\xc4\x00B\x00\xe8\x00C\x00'
>>> S = 'spam'
>>> S.encode('utf-16')
b'\xff\xfe\x00p\x00a\x00m\x00'
>>> S.encode('utf-32')
b'\xff\xfe\x00\x00s\x00\x00\x00p\x00\x00\x00a\x00\x00\x00m\x00\x00\x00'

```

Literały bajtowe

Dwie uwagi. Po pierwsze, Python 3.x pozwala na kodowanie znaków specjalnych za pomocą sekwencji ucieczki szesnastkowych i Unicode w łańcuchach znaków `str`, jednak w łańcuchach `bytes` tylko z użyciem wersji szesnastkowej. Sekwencje ucieczki Unicode są po cichu uznawane za dosłowne literały obiektów `bytes`, a nie sekwencje ucieczki. Tak naprawdę w celu poprawnego wyświetlenia znaków spoza zakresu ASCII obiekty `bytes` trzeba zdekodować do łańcuchów `str`:

```

>>> S = 'A\xC4B\xE8C'                                # str rozpoznaje sekwencje ucieczki
Unicode i szesnastkowe

>>> S
'AÄBèC'

>>> S = 'A\u00C4B\U000000E8C'
>>> S
'AÄBèC'

>>> B = b'A\xC4B\xE8C'                            # bytes rozpoznaje szesnastkowe
sekwencje ucieczki, Unicode nie

>>> B
b'A\xc4B\xe8C'

>>> B = b'A\u00C4B\U000000E8C'                  # Sekwencje ucieczki potraktowane
dosłownie!

>>> B
b'A\ 00C4B\ 000000E8C'

>>> B = b'A\xC4B\xE8C'                            # W przypadku bytes trzeba użyć
szesnastkowych sekwencji ucieczki

>>> B
# Wyświetla znaki spoza ASCII jako

b'A\xc4B\xe8C'

>>> print(B)
b'A\xc4B\xe8C'

>>> B.decode('latin-1')                           # Dekoduje jako latin-1, by
zinterpretować jako tekst

```

```
'AÄBèC'
```

Po drugie, literały łańcuchów bytes wymagają, by znaki były albo z zakresu ASCII, albo — jeśli ich wartość jest większa od 127 — by były opatrzone znakami ucieczki. Łańcuchy str z kolei zezwalają na stosowanie w literałach dowolnych znaków ze źródłowego zbioru znaków (którym, jak omówimy to później, domyślnie jest UTF-8 w wersji 3.x i ASCII w 2.x, o ile w pliku źródłowym nie ma deklaracji kodowania):

```
>>> S = 'AÄBèC'                                # Znaki z UTF-8, jeśli nie ma deklaracji
                                                kodowania

>>> S
'AÄBèC'

>>> B = b'AÄBèC'
SyntaxError: bytes can only contain ASCII literal characters.

>>> B = b'A\xC4B\xE8C'                      # Znaki muszą być ASCII lub z sekwencjami
                                                ucieczki

>>> B
b'A\xc4B\xe8C'

>>> B.decode('latin-1')
'AÄBèC'

>>> S.encode()                                 # Kod źródłowy domyślnie zakodowany jako
                                                UTF-8

b'A\xc3\x84B\xc3\xa8C'                      # Wykorzystuje ustawienie domyślne systemu
                                                do zdekodowania,
                                                # o ile nie przekazano kodowania

>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'

>>> B.decode()                               # Surowe bajty nie odpowiadają UTF-8
                                               
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: ...
```

Należy pamiętać, że literał bajtowy zawiera dane binarne, a nie znaki o podanych kodach Unicode. Taki literał może zawierać tekst, jednak generalnie bajty nie reprezentują kodów znaków, chyba że znaki te wcześniej zostały zdekodowane.

Konwersja kodowania

Dotychczas kodowaliśmy i dekodowaliśmy łańcuchy znaków w celu zbadania ich struktury. Mówiąc bardziej ogólnie, możemy zawsze przekształcić łańcuch znaków na kodowanie inne od ustawienia domyślnego zbioru znaków źródłowych, jednak musimy w jawnym sposób podać nazwę kodowania. Dotyczy to szczególnie przypadku, w którym źródłowy tekst jest odczytywany z pliku lub zdefiniowany jako literał.

Termin *konwersja* może być nieco mylący, ponieważ w rzeczywistości chodzi tu o kodowanie tekstu w postaci surowych bajtów według innego schematu niż oryginalny. Jak pisałem wcześniej, ciąg zapisany w pamięci komputera nie jest kodowany według żadnego schematu. Jest to po prostu sekwencja kodów Unicode (czyli znaków), zatem pojęcie zmiany kodowania tutaj nie istnieje. Można jednak tworzyć skrypty odczytujące ciągi zakodowane w jednym

formacie i zapisujące je w innym. Dzięki temu te same dane mogą być wykorzystywane przez różne programy.

```
>>> B = b'A\xc3\x84B\xc3\xa8C'          # Oryginalny tekst zakodowany w formacie  
UTF-8  
>>> S = B.decode('utf-8')                # Zdekodowany tekst w formacie UTF-8  
>>> S  
'AÄBèC'  
>>> T = S.encode('cp500')                # Konwersja na EBCDIC  
>>> T  
b'\xc1c\xc2T\xc3'  
>>> U = T.decode('cp500')                # Konwersja z powrotem na Unicode  
>>> U  
'AÄBèC'  
>>> U.encode()                         # Znowu domyślne kodowanie UTF-8  
b'A\xc3\x84B\xc3\xa8C'
```

Należy pamiętać, że specjalne sekwencje ucieczki Unicode i szesnastkowe są niezbędne jedynie wówczas, kiedy ręcznie piszemy kod łańcuchów znaków Unicode spoza ASCII. W praktyce często zamiast tego taki tekst ładujemy z plików. Jak zobaczymy później w niniejszym rozdziale, obiekt pliku z Pythona 3.x (tworzony za pomocą wbudowanej funkcji open) automatycznie dekoduje tekstowe łańcuchy znaków w miarę ich wczytywania i koduje je w czasie zapisywania. Z tego powodu skrypty mogą często obsługiwać łańcuchy znaków w sposób ogólny, bez konieczności bezpośredniego zapisywania w kodzie znaków specjalnych.

W dalszej części rozdziału zobaczymy również, że można także przekształcać tekst z jednego kodowania na inne przy przenoszeniu łańcuchów znaków z plików i do nich, wykorzystując do tego technikę bardzo podobną do tej z ostatniego przykładu. Choć nadal będziemy musieli w jawny sposób udostępnić nazwy kodowania przy otwieraniu pliku, interfejs plików wykonuje większość zadań związanych z konwersją w sposób automatyczny.

Łańcuchy znaków Unicode w Pythonie 2.x

Dużo piszę o ciągach Unicode w Pythonie 3.x, ponieważ jest to nowość. Skoro już zaprezentowałem podstawy łańcuchów znaków Unicode z Pythona 3.x, powiniem wyjaśnić, że prawie to samo można uzyskać w wersji 2.x, choć za pomocą innych narzędzi. Typ unicode dostępny jest w Pythonie 2.x, jednak jest typem danych odrębnym od str i pozwala na swobodne mieszanie zwykłych łańcuchów znaków i łańcuchów Unicode, kiedy są one ze sobą zgodne.

Tak naprawdę możemy udawać, że typ str z wersji 2.x jest typem bytes z wersji 3.x, jeśli chodzi o dekodowanie surowych bajtów na łańcuchy Unicode, o ile są one w poprawnej formie. Poniżej znajduje się przykład działania Pythona 2.x. Znaki unicode wyświetlane są w wersji 2.x jako szesnastkowe, o ile ich w jawny sposób nie wyświetlimy, natomiast wyświetlanie znaków spoza ASCII wygląda różnie w różnych powłokach. (Wyniki większości przykładów z tego podrozdziału będą inne w środowisku IDLE, które niekiedy wykrywa standardowe kodowania i wyświetla znaki Latin-1 w postaci kodów bajtów. Więcej na ten temat można dowiedzieć się z opisu zmiennej środowiskowej PYTHONIOENCODING i sposobu wyświetlania znaków w wierszu poleceń Windows).

```
C:\code> C:\python27\python
>>> S = 'A\xC4B\xE8C'                      # Ciąg 8-bitowych wartości
>>> S                                     # Tekst zakodowany w standardzie Latin-1,
zawierający znaki spoza ASCII
'A\xc4B\xe8C'

>>> print S                                # Niedrukowalne znaki (w środowisku IDLE
mogą wyglądać inaczej)
A□B□C

>>> S.decode('latin-1')                     # Zdekodowanie bajta do Unicode w Latin-1
u'A\xc4B\xe8C'

>>> S.decode('utf-8')                       # Nie jest sformatowane jak UTF-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1: invalid
continuation byte

>>> S.decode('ascii')                      # Poza zakresem ASCII
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1:
ordinal not in range(128)
```

By przechować dowolnie zakodowany tekst Unicode, należy utworzyć obiekt `unicode` za pomocą literała w postaci `u'xxx'` (literał ten nie jest już dostępny w wersji 3.x, ponieważ wszystkie łańcuchy znaków obsługują w tej wersji Unicode):

```
>>> U = u'A\xC4B\xE8C'                  # Utworzenie łańcucha Unicode,
szesnastkowe sekwencje ucieczki

>>> U
u'A\xc4B\xe8C'

>>> print U
AÄBèC
```

Po utworzeniu możemy przekształcić tekst Unicode na różne typy kodowania surowych bajtów, podobnie do kodowania obiektów `str` na obiekty `bytes` w Pythonie 3.x:

```
>>> U.encode('latin-1')                 # Kodowanie Latin-1: 8-bitowe bajty
'A\xc4B\xe8C'

>>> U.encode('utf-8')                  # Kodowanie UTF-8: wielobajtowe
'A\xc3\x84B\xc3\x8a8C'
```

Znaki spoza zakresu ASCII można zapisywać w Pythonie 2.x w kodzie literałów łańcuchów znaków za pomocą sekwencji ucieczki szesnastkowych bądź Unicode, tak samo jak w wersji 3.x. Tak samo jednak jak w przypadku typu `bytes` w 3.x, w wersji 2.x sekwencje ucieczki "`\u...`" oraz "`\U...`" rozpoznawane są jedynie w przypadku łańcuchów znaków `unicode`, a nie 8-bitowych łańcuchów `str`:

```
C:\code> c:\python27\python
>>> U = u'A\xC4B\xE8C'                  # Szesnastkowe sekwencje ucieczki dla
znaków spoza ASCII

>>> U
```

```

u'A\xc4B\xe8C'

>>> print U
AÄBèC

>>> U = u'A\u00C4B\U000000E8C'          # Sekwencje ucieczki Unicode dla znaków
spoza ASCII

>>> U                                # u'' = 16 bitów, U'' = 32 bity
u'A\xc4B\xe8C'

>>> print U
AÄBèC

>>> S = 'A\xC4B\xE8C'                 # Szesnastkowe sekwencje ucieczki
działają

>>> S
'A\xc4B\xe8C'

>>> print S                          # Niektóre są dziwnie wyświetlane, jeśli
się ich nie zdekoduje
A□B□FC

>>> print S.decode('latin-1')
AÄBèC

>>> S = 'A\u00C4B\U000000E8C'          # Nie sekwencje ucieczki Unicode –
traktowane są dosłownie!

>>> S
'A\u00C4B\U000000E8C'

>>> print S
A\u00C4B\U000000E8C

>>> len(S)
19

```

Mieszanie typów ciągów w wersji 2.x

Tak jak typy `str` i `bytes` z Pythona 3.x, typy `unicode` i `str` z wersji 2.x współdzielą prawie identyczny zbiór operacji, dlatego o ile nie musimy przekształcać ich na inne kodowanie, często możemy traktować obiekty `unicode` tak, jakby były obiektami `str`. Jedną z głównych różnic między 2.x a 3.x jest jednak to, że `unicode` i obiekty `str` poza Unicode można ze sobą swobodnie *mieszać* w wyrażeniach, a dopóki obiekt `str` jest zgodny z kodowaniem `unicode`, Python automatycznie przekształci go na `unicode`.

```

>>> u'ab' + 'cd'                      # Mogą się w 2.x mieszkać, jeśli są zgodne
u'abcd'                               # 'ab' + b'cd' nie jest dozwolone w wersji
3.x

```

Tak liberalne podejście do mieszania typów łańcuchów znaków w Pythonie 2.x działa jednak tylko wtedy, gdy串bajtów zawiera jedynie 7-bitowe kody ASCII:

```

>>> S = 'A\xC4B\xE8C'          # W wersji 2.x nie można mieszać, jeśli
ciąg zawiera znaki spoza ASCII!
>>> U = u'A\xC4B\xE8C'
>>> S + U

UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1:
ordinal not in range(128)

>>> 'abc' + U                # Można mieszać jedynie wtedy, gdy
wszystkie znaki są 7-bitowymi kodami ASCII
u'abcA\xc4B\xe8C'

>>> print 'abc' + U          # Aby wyświetlić znaki, należy użyć
instrukcji print
abcAÄBèC

>>> S.decode('latin-1') + U    # W wersji 2.x może być wymagana jawnia
konwersja
u'A\xc4B\xe8CA\xc4B\xe8C'

>>> print S.decode('latin-1') + U
AÄBèCAÄBèC

>>> print u'\xA3' + '999.99'    # Patrz również przykład z walutami w
rozdziale 25.

£999.99

```

Natomiast w wersji 3.x nie można mieszać typów `str` i `bytes`, dlatego w takich wypadkach wymagana jest jawnia konwersja. Powyższy kod został wprawdzie wykonany poprawnie w wersji 3.3, ale tylko dlatego, że literał Unicode w wersji 2.x jest taki sam jak typ `str` w wersji 3.x (przy czym znak `u` jest pomijany). Analogiczną operacją w wersji 3.x byłoby łączenie ciągów typu `str` i `bytes` (np. `'ab' + b'cd'`), które jest błędne, chyba że oba ciągi można przekształcić do wspólnego typu.

W wersji 2.x różnice między typami są często nieistotne. Ciągi Unicode, tak jak zwykłe ciągi, można łączyć, indeksować, dzielić, przetwarzać za pomocą modułu `React`, ale nie można ich modyfikować. Jeżeli trzeba jawnie zmienić typ ciągu, należy użyć wbudowanej funkcji `str` lub `unicode`, jak niżej:

```

>>> str(u'mielonka')          # Konwersja ciągu Unicode na zwykły ciąg
'mielonka'
>>> unicode('mielonka')       # Konwersja zwykłego ciągu na Unicode
u'mielonka'

```

Wreszcie, jak zobaczymy bardziej szczegółowo później w niniejszym rozdziale, wywołanie `open` z Pythona 2.x obsługuje jedynie pliki bajtów 8-bitowych, zwracając ich zawartość jako łańcuchy znaków `str`. To od nas zależy interpretacja zawartości jako tekstu lub danych binarnych i ewentualne zdekodowanie. By wczytać i zapisać pliki Unicode, a także automatycznie zakodować lub zdekodować ich zawartość, należy użyć wywołania `codecs.open` z Pythona 2.x, opisanego w dalszej części rozdziału. Wywołanie to udostępnia prawie tę samą funkcjonalność co `open` z wersji 3.x i wykorzystuje obiekty `unicode` z 2.x do reprezentowania zawartości pliku. Wczytanie pliku przekłada zakodowane bajty na zdekodowane znaki Unicode, a zapisywanie przekłada łańcuchy znaków na pożądane kodowanie określone przy otwieraniu pliku.

Deklaracje typu kodowania znaków pliku źródłowego

Sekwencje ucieczki Unicode są w porządku w przypadku okazjnego zastosowania znaku Unicode w literale łańcucha znaków, jednak ich użycie może stać się żmudne, jeśli często musimy osadzać tekst spoza zakresu ASCII w łańcuchach znaków. W przypadku łańcuchów znaków zapisywanych w kodzie plików skryptów Python domyślnie wykorzystuje kodowanie UTF-8, jednak pozwala je zmienić tak, by obsługiwać dowolny typ kodowania, umieszczając komentarz z nazwami pożądanego kodowania. Komentarz ten musi mieć następującą postać i musi się pojawiać w pierwszym lub drugim wierszu skryptu zarówno w Pythonie 2.x, jak i 3.x:

```
# -*- coding: latin-1 -*-
```

Kiedy komentarz w tej formie jest obecny w pliku, Python rozpozna łańcuchy znaków reprezentowane w podanym kodowaniu. Oznacza to, że możemy edytować plik skryptu w edytorze tekstu przyjmującym i poprawnie wyświetlającym znaki akcentowane i inne znaki spoza zakresu ASCII, a Python poprawnie je zdekoduje w literałach łańcuchów znaków. Przykładowo warto zwrócić uwagę, jak komentarz umieszczony na górze poniższego pliku *text.py* pozwala na osadzanie znaków Latin-1 w łańcuchach znaków:

```
# -*- coding: latin-1 -*-

# Każda z poniższych form literału łańcucha znaków działa w latin-1.

# Zmiana kodowania wyżej na ASCII albo UTF-8 nie powiedzie się,
# ponieważ znaki 0xc4 i 0xe8 z myStr1 nie są w nich poprawne.

myStr1 = 'aÄBèC'

myStr2 = 'A\u00c4B\u00e8C'

myStr3 = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'

import sys

print('Kodowanie systemowe:', sys.getdefaultencoding())

for aStr in myStr1, myStr2, myStr3:

    print('{0}, strlen={1}'.format(aStr, len(aStr)), end='')

    bytes1 = aStr.encode()                      # Zgodnie z domyślnym UTF-8: 2 bajty
    dla znaków spoza ASCII

    bytes2 = aStr.encode('latin-1')              # Jeden bajt na znak

    #bytes3 = aStr.encode('ascii')                # ASCII nie działa: poza zakresem
    0..127

    print('byteslen1={0}, byteslen2={1}'.format(len(bytes1), len(bytes2)))
```

Po wykonaniu powyższy skrypt daje poniższe wyniki dla trzech ciągów uzyskanych różnymi technikami. Wyświetlane są ciągi znaków, ich długości oraz liczby bajtów w standardach UTF-8 i Latin-1:

```
C:\code> c:\python33\python text.py
Kodowanie systemowe: utf-8
aÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
```

Ponieważ większość programistów będzie polegała na standardowym kodowaniu UTF-8, osoby zainteresowane szczegółowymi informacjami dotyczącymi tej opcji, a także innymi zaawansowanymi zagadnieniami związanymi z Unicode, takimi jak właściwości i sekwencje ucieczki znaków w łańcuchach, odsyłam do dokumentacji biblioteki standardowej Pythona. W tym rozdziale przyjrzymy się bliżej nowemu obiektowi tekstowemu wprowadzonemu w wersji 3.x, a następnie zajmiemy się różnicami w narzędziach i operacjach na plikach.



Dodatkowym przykładem kodowania znaków spoza zestawu ASCII oraz deklarowania standardu kodowania plików źródłowych jest program, w którym do formatowania kwoty został wykorzystany symbol waluty. Znajduje się on w rozdziale 25. i załączonym do książki pliku *formats_currency2.py*. W tym pliku niezbędne było użycie deklaracji standardu kodowania, ponieważ w programie zostały użyte symbole waluty będące znakami spoza zestawu ASCII. Przykład ten pokazuje jednocześnie możliwość przenoszenia kodu zawierającego literały Unicode z wersji 2.x do 3.x.

Wykorzystywanie obiektów bytes z Pythona 3.x

W rozdziale 7. omówiliśmy szeroki zakres operacji dostępnych w ogólnym typie łańcucha znaków Pythona 3.x `str`. Podstawowy typ danych łańcucha znaków działa w wersjach 2.x oraz 3.x identycznie, dlatego nie będziemy wracać do tego zagadnienia. Zamiast tego zagłębimy się nieco w zbiór operacji udostępnianych przez nowy typ `bytes` z Pythona 3.x.

Jak wspomniano wcześniej, obiekt `bytes` z wersji 3.x jest sekwencją niewielkich liczb całkowitych, z których każda mieści się w przedziale od 0 do 255; zaprojektowano go tak, by po wyświetleniu pokazywał znaki ASCII. Obsługuje on operacje na sekwencjach i większość metod dostępnych dla obiektów `str` (a także obecnych w typie `str` z Pythona 2.x). Obiekt `bytes` *nie* obsługuje jednak metody `format` ani wyrażenia formatującego ze znakiem `%`. Nie można także mieszać i dopasowywać obiektów `bytes` i `str` bez jawnego przekształcenia ich. Zazwyczaj obiekty typu `str` wykorzystuje się w plikach tekstowych z przeznaczeniem na *dane tekstowe*, natomiast obiekty typu `bytes` służą w plikach binarnych do reprezentowania *danych binarnych*.

Wywołania metod

Jeśli naprawdę chcemy zobaczyć, jakie atrybuty, których nie ma typ `bytes`, znajdują się w typie `str`, zawsze możemy sprawdzić wyniki wywołania funkcji wbudowanej `dir`. Jej dane wyjściowe powiedzą nam także coś o obsługiwanych operatorach wyrażeń (przykładowo `__mod__` oraz `__rmod__` implementują operator `%`).

```
C:\code> c:\python33\python
# Atrybuty unikalne dla typu str
>>> set(dir('abc')) - set(dir(b'abc'))
{'isdecimal', '__mod__', '__rmod__', 'format_map', 'isprintable',
'casemap', 'format', 'isnumeric', 'isidentifier', 'encode'}
# Atrybuty unikalne dla typu bytes
>>> set(dir(b'abc')) - set(dir('abc'))
```

```
{'decode', 'fromhex'}
```

Jak widać, typy `str` i `bytes` mają prawie identyczną funkcjonalność. Ich unikalne atrybuty to zazwyczaj metody, które nie mają zastosowania do tego drugiego typu. Przykładowo metoda `decode` przekłada surowe bajty na ich reprezentację `str`, natomiast `encode` tłumaczy łańcuch znaków na jego reprezentację w postaci surowych bajtów. Większość metod jest taka sama, choć metody `bytes` wymagają przekazania argumentów w postaci obiektów `bytes` (znów: typy łańcuchów znaków w Pythonie 3.x nie mieszają się ze sobą). Warto także przypomnieć, że obiekty typu `bytes` są niezmienne, tak samo jak obiekty `str`, zarówno w wersji 2.x, jak i 3.x (komunikaty o błędach zostały skrócone w imię oszczędności miejsca).

```
>>> B = b'mielonka'                      # Literał obiektu bytes b'...'  
>>> B.find(b'ie')  
1  
>>> B.replace(b'ie', b'XY')            # Metody bytes oczekują argumentów bytes  
b'mXYlonka'  
>>> B.split(b'ie')  
[b'm', b'lonka']  
>>> B  
b'mielonka'  
>>> B[0] = 'x'  
TypeError: 'bytes' object does not support item assignment
```

Jedyną istotną różnicą jest to, że *formatowanie łańcuchów znaków* działa w Pythonie 3.x jedynie na obiektach `str`, natomiast na obiektach `bytes` już nie (więcej informacji na temat wyrażeń i metod formatowania łańcuchów znaków znajduje się w rozdziale 7.):

```
>>> '%s' % 99  
'99'  
>>> b'%s' % 99  
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'  
>>> '{0}'.format(99)  
'99'  
>>> b'{0}'.format(99)  
AttributeError: 'bytes' object has no attribute 'format'
```

Operacje na sekwencjach

Oprócz wywołań metod wszystkie znane nam (i przez nas kochane) ogólne operacje na sekwencjach z łańcuchów znaków i list Pythona 2.x działają zgodnie z oczekiwaniami na obiektach `str` oraz `bytes` w wersji 3.x. Obejmuje to między innymi indeksowanie, wycinki czy konkatenację. Warto zwrócić uwagę na to, jak w poniższym kodzie indeksowanie obiektu `bytes` zwraca liczbę całkowitą podającą wartość binarną bajta. Obiekt `bytes` jest tak naprawdę *sekwencją 8-bitowych liczb całkowitych*, jednak gdy jest wyświetlany w całości, dla wygody prezentuje się jako łańcuch znaków w kodzie ASCII. By sprawdzić wartość podanego bajta,

należy użyć funkcji wbudowanej `chr` w celu przekształcenia jej z powrotem na znak — jak w poniższym kodzie:

```
>>> B = b'mielonka'                      # Sekwencja niewielkich liczb całkowitych
>>> B                                     # Wyświetlana jako znaki ASCII
b'mielonka'
>>> B[0]                                  # Indeksowanie zwraca liczbę całkowitą
109
>>> B[-1]
97
>>> chr(B[0])                            # Pokazanie znaku dla liczby całkowitej
'm'
>>> list(B)                               # Pokazanie wszystkich wartości liczb
całkowitych obiektu bytes
[109, 105, 101, 108, 111, 110, 107, 97]
>>> B[1:], B[:-1]
(b'ielonka', b'mielonk')
>>> len(B)
8
>>> B + b'lmn'
b'mielonkalmn'
>>> B * 4
b'mielonkamielonkamielonkamielonka'
```

Inne sposoby tworzenia obiektów bytes

Dotychczas większość obiektów `bytes` tworzyliśmy za pomocą składni literała `'b...'`. Można je jednak tworzyć także za pomocą wywołania konstruktora `bytes` z obiektem `str` i nazwą kodowania, wywołania konstruktora `bytes` z obiektem liczb całkowitych reprezentującym wartości bajtowe, na którym można wykonywać iterację, a także kodując obiekt `str` zgodnie z domyślnym (lub przekazanym) typem kodowania. Jak widzieliśmy, operacja kodowania pobiera obiekt `str` i zwraca wartość bajtową łańcucha znaków zgodnie ze specyfikacją tego kodowania. I odwrotnie, operacja dekodowania pobiera sekwencję surowych bajtów i koduje ją na jej reprezentację łańcucha znaków — serię znaków. Oba działania tworzą nowe obiekty łańcucha znaków.

```
>>> B = b'abc'                           # Literał
>>> B
b'abc'
>>> B = bytes('abc', 'ascii')           # Wywołanie konstruktora z nazwą kodowania w
argumencie
>>> B
```

```

b'abc'
>>> ord('a')
97
>>> B = bytes([97, 98, 99])      # Lista liczb całkowitych
>>> B
b'abc'
>>> B = 'mielonka'.encode()      # Metoda str.encode lub bytes
>>> B
b'mielonka'
>>>
>>> S = B.decode()              # Metoda bytes.decode lub str
>>> S
'mielonka'

```

Z szerszej perspektywy dwie ostatnie z przedstawionych operacji są tak naprawdę narzędziami służącymi do *konwersji* pomiędzy obiektami `str` i `bytes`, czyli tematu wprowadzonego wcześniej i rozszerzonego w kolejnym podrozdziale.

Mieszanie typów łańcuchów znaków

W wywołaniu z `replace` w podrozdziale „Wywołania metod” musieliśmy przekazać dwa obiekty `bytes` — obiekty `str` by tam nie zadziałyły. Choć Python 2.x dokonuje automatycznej konwersji między typami `str` i `unicode`, gdy jest to możliwe (to jest gdy `str` jest 7-bitowym tekstem ASCII), Python 3.x wymaga w pewnych kontekstach określonych typów łańcuchów znaków, a kiedy jest to potrzebne — ręcznych konwersji:

```

# Musi przekazywać oczekiwane typy do wywołań funkcji oraz metod
>>> B = b'mielonka'
>>> B.replace('ie', 'XY')
TypeError: expected an object with the buffer interface
>>> B.replace(b'ie', b'XY')
b'mXYlonka'
>>> B = B'mielonka'
>>> B.replace(bytes('ie'), bytes('xy'))
TypeError: string argument without an encoding
>>> B.replace(bytes('ie', 'ascii'), bytes('xy', 'utf-8'))
b'mxylonka'
# Musi dokonywać ręcznej konwersji w wyrażeniach z mieszanymi typami
>>> b'ab' + 'cd'
TypeError: can't concat bytes to str

```

```

>>> b'ab'.decode() + 'cd'           # Z bytes na str
'abcd'
>>> b'ab' + 'cd'.encode()         # Ze str na bytes
b'abcd'
>>> b'ab' + bytes('cd', 'ascii')  # Ze str na bytes
b'abcd'

```

Choć można samemu tworzyć obiekty `bytes` w celu reprezentowania spakowanych danych binarnych, są one także tworzone automatycznie po wczytaniu plików otwartych w trybie binarnym, co zobaczymy w szczegółach nieco później w niniejszym rozdziale. Najpierw jednak powinniśmy zapoznać się z bardzo bliskim i zmiennym krewnym obiektu `bytes`.

Obiekt bytearray w wersji 3.x (oraz 2.6 lub nowszej)

Dotychczas koncentrowaliśmy się na typach `str` i `bytes`, ponieważ można je podciągnąć pod typy `unicode` i `str` z Pythona 2.x. Python 3.x udostępnia jednak także trzeci typ łańcucha znaków — `bytearray`, czyli zmienna sekwencja liczb całkowitych w przedziale od 0 do 255, która jest zasadniczo zmiennym wariantem typu `bytes`. Tym samym obsługuje te same metody łańcuchów znaków i działania na sekwencjach co `bytes`, a także wiele z działań modyfikacji w miejscu obsługiwanych przez `listy`.

Znaki w ciągach typu `bytearray` mogą być zamieniane na typowe dane binarne, jak również na proste ciągi, np. ASCII, w których każdy bajt reprezentuje jeden znak (bogatsze ciągi Unicode wymagają innego kodowania, a poza tym są niemutowalne). Typ `bytearray` jest dostępny również w Pythonie 2.6 i 2.7 jako przeniesienie z wersji 3.x, jednak nie wymusza tam ścisłego rozróżnienia między danymi tekstowymi a binarnymi z Pythona 3.x.

Typ bytearray w akcji

Zajmijmy się teraz szybkim omówieniem tego typu. Obiekty `bytearray` można tworzyć, wywołując funkcję wbudowaną `bytearray`. W Pythonie do inicjalizacji można wykorzystać dowolny łańcuch znaków:

```

# Utworzenie w Pythonie 2.6/2.7: zmienna sekwencja małych (0..255) liczb
# całkowitych

>>> S = 'mielonka'

>>> C = bytearray(S)               # Przeniesienie z wersji 3.x do 2.6
lub nowszej

>>> C                           # b'...' == '...' w wersji 2.6 lub
nowszej (str)

bytearray(b'mielonka')

```

W Pythonie 3.x wymagana jest nazwa kodowania lub łańcuch bajtowy, ponieważ łańcuchy tekstowe i binarne nie mieszają się ze sobą, choć łańcuchy bajtowe mogą odzwierciedlać tekst zakodowany w Unicode:

```
# Utworzenie w Pythonie 2.x: tekst i dane binarne nie mieszają się ze sobą
```

```

>>> S = 'mielonka'
>>> C = bytearray(S)
TypeError: string argument without an encoding
>>> C = bytearray(S, 'latin-1')          # Typ odpowiedni dla zawartości w
wersji 3.x
>>> C
bytearray(b'mielonka')
>>> B = b'mielonka'                   # b'...' != '...' w wersji 3.x
(bytes/str)
>>> C = bytearray(B)
>>> C
bytearray(b'mielonka')

```

Po utworzeniu obiekty `bytearray` stają się sekwencjami małych liczb całkowitych, tak jak typ `bytes`, i są zmienne, tak jak listy, choć dla operacji przypisania do indeksu wymagają liczb całkowitych, a nie łańcuchów znaków. Poniższy listing jest w całości kontynuacją powyższej sesji interaktywnej i wykonywany jest w Pythonie 3.x, o ile nie jest zaznaczone inaczej — komentarze dotyczące wersji 2.x znajdują się w uwagach.

```

# Obiekt zmienny, jednak musi przypisywać liczby całkowite, a nie łańcuchy
znaków
>>> C[0]
109
>>> C[0] = 'x'                      # Ta i poniższa operacja działają w
wersji 2.6/2.7
TypeError: an integer is required
>>> C[0] = b'x'
TypeError: an integer is required
>>> C[0] = ord('x')                 # Funkcja ord zwraca kod znaku
>>> C
bytearray(b'xielonka')
>>> C[1] = b'Y'[0]                  # Do poszczególnych bajtów można
odwoływać się za pomocą indeksu
>>> C
bytearray(b'xYelonka')

```

Przetwarzanie obiektów `bytearray` zapożycza zarówno z łańcuchów znaków, jak i list, ponieważ obiekty te są zmiennymi łańcuchami bajtów. Obiekt `bytearray` ma wiele metod takich samych, jakie mają obiekty `str` i `bytes`, a dodatkowo sporo modyfikujących zawartość metod, takich jak w obiekcie `list`. Oprócz nazwanych metod metody `__iadd__` oraz `__setitem__` obiektu `bytearray` implementują, odpowiednio, konkatenację w miejscu `+=` oraz przypisanie do indeksu.

```
# Metody dostępne w obiekcie bytes, ale nie w bytearray
```

```
>>> set(dir(b'abc')) - set(dir(bytarray(b'abc')))  
{'__getnewargs__'}  
# Metody dostępne w obiekcie bytarray, ale nie w bytes  
  
>>> set(dir(bytarray(b'abc'))) - set(dir(b'abc'))  
{'__iadd__', 'reverse', '__setitem__', 'extend', 'copy', '__alloc__',  
 '__delitem__', '__imul__', 'remove', 'clear', 'insert', 'append', 'pop'}
```

Obiekt `bytearray` można zmodyfikować w miejscu za pomocą zarówno przypisania do indeksu, co właśnie widzieliśmy, jak i zaprezentowanych niżej metod podobnych do list. By zmodyfikować tekst w miejscu, w wersji 2.6 musielibyśmy przekształcić go na listę i z powrotem za pomocą `list(str)` i `'.join(list)` (patrz przykłady w rozdziałach 4. i 6.):

```
# Wywołania zmiennych metod

>>> C
bytearray(b'xYelonka')

>>> C.append(b'MN')                                # Python 2.x wymaga łańcucha znaków o
wielkości 1

TypeError: an integer is required

>>> C.append(ord('L'))

>>> C
bytearray(b'xYelonkaL')

>>> C.extend(b'MNO')

>>> C
bytearray(b'xYelonkaLMNO')
```

Na obiektach `bytearray` zgodnie z oczekiwaniemi działają wszystkie zwykłe operacje na sekwencjach oraz metody łańcuchów znaków (warto zauważyć, że tak jak w przypadku typu `bytes`, wyrażenia i metody oczekują argumentów będących obiektami `bytes`, a nie `str`).

```
# Operacje na sekwencjach i metody łańcuchów znaków

>>> C
bytearray(b'xYelonkaLMNO')

>>> C + b'!#'
bytearray(b'xYelonkaLMNO!#')

>>> C[0]
120

>>> C[1:]
bytearray(b'YelonkaLMNO')

>>> len(C)
12

>>> C.replace('xY', 'mi') # Ten kod działa w wersji 2.x
```

```
TypeError: Type str doesn't support the buffer API
>>> C.replace(b'xY', b'mi')
bytearray(b'mielonkaLMNO')
>>> C
bytearray(b'xYelonkaLMNO')
>>> C * 4
bytearray(b'xYelonkaLMNOxYelonkaLMNOxYelonkaLMNOxYelonkaLMNO')
```

Podsumowanie typów ciągów znaków w Pythonie 3.x

Wreszcie, słowem podsumowania, poniższe przykłady demonstrują, że obiekty `bytes` oraz `bytearray` są sekwencjami liczb całkowitych, natomiast obiekty `str` — sekwencjami znaków.

```
# Dane binarne a dane tekstowe
>>> B                         # B jest takie samo jak S w wersji 2.6/2.7
b'mielonka'
>>> list(B)
[109, 105, 101, 108, 111, 110, 107, 97]
>>> C
bytearray(b'xYelonkaLMNO')
>>> list(C)
[120, 89, 101, 108, 111, 110, 107, 97, 76, 77, 78, 79]
>>> S
'mielonka'
>>> list(S)
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a']
```

Choć wszystkie trzy typy łańcuchów znaków z Pythona 3.x zawierają wartości będące znakami i obsługują wiele z tych samych operacji, należy pamiętać, by zawsze:

- używać typu `str` dla danych tekstowych,
- używać typu `bytes` dla danych binarnych,
- używać typu `bytearray` dla danych binarnych, które chcemy modyfikować w miejscu.

Powiązane narzędzia, takie jak pliki, czyli temat kolejnego podrozdziału, często decydują o tym za nas.

Wykorzystywanie plików tekstowych i binarnych

Niniejszy rozdział rozszerza zagadnienia związane z wpływem modelu łańcuchów znaków z Pythona 3.x na przetwarzanie plików, którego podstawy omówiliśmy we wcześniejszej części książki. Jak wspomniano wcześniej, kluczowy jest tu tryb otwierania pliku — określa on, jaki typ obiektu zostanie wykorzystany do reprezentowania w skrypcie zawartości pliku. Tryb tekstowy narzuca obiekty `str`, natomiast tryb binarny — obiekty `bytes`.

- *Pliki w trybie tekstowym* interpretują zawartość zgodnie z *kodowaniem Unicode* — albo zgodnie z domyślnym ustawieniem platformy, albo według przekazanej nazwy kodowania. Przekazując do `open` nazwę kodowania, możemy wymusić konwersję na różne typy plików Unicode. Pliki w trybie tekstowym wykonują również uniwersalne *translacje końca wiersza*. Domyślnie wszystkie formy końca wiersza odwzorowywane są w skrypcie na pojedynczy znak '`\n`', bez względu na platformę, na której skrypt ten wykonujemy. Jak wspomniano wcześniej, pliki tekstowe obsługują również odczytywanie i zapisywanie *znacznika kolejności bajtów* (BOM) przechowywanego na początku pliku w niektórych schematach kodowania Unicode.
- *Pliki w trybie binarnym* zwracają zamiast tego zawartość w postaci *surowej*, jako sekwencję liczb całkowitych reprezentujących wartości bajtowe, bez kodowania i dekodowania, a także bez translacji końca wiersza.

Drugi argument przekazany do funkcji `open` określa, czy chcemy plik przetwarzać jako tekstowy, czy binarny, tak samo jak w Pythonie 2.x. Dodanie `b` do tego łańcucha wymusza tryb binarny, jak na przykład w "`rb`" wczytującym binarne pliki z danymi. Trybem domyślnym jest "`rt`" — jest to to samo co "`r`" i oznacza tekstowe dane wejściowe (tak samo jak w wersji 2.x).

W Pythonie 3.x ten argument funkcji `open` określa także *typ obiektu* dla reprezentacji zawartości pliku, bez względu na platformę. Pliki tekstowe zwracają obiekt `str` dla odczytu i oczekują tego obiektu dla zapisu, natomiast pliki binarne zwracają obiekt `bytes` dla odczytu i oczekują tego samego (bądź obiektu `bytearray`) dla zapisu.

Podstawy plików tekstowych

W celach demonstracyjnych zaczniemy od prostego pliku wejścia-wyjścia. Dopóki będziemy przetwarzać proste pliki tekstowe (na przykład ASCII), nie będziemy się przejmować obchodzeniem domyślnego kodowania łańcuchów znaków platformy. Pliki w Pythonie 3.x wyglądają i działają w dużej mierze tak samo jak w 2.x (w tym zakresie podobnie zachowują się łańcuchy znaków). Poniższy kod zapisuje na przykład jeden wiersz tekstu do pliku i wczytuje go z powrotem w wersji 3.x. Dokładnie tak samo działałyby to w wersji 2.6 (warto zauważyć, że `file` w Pythonie 3.x nie jest już nazwą wbudowaną, dlatego można jej użyć w kodzie jako zmiennej).

```
C:\code> c:\python33\python

# Proste pliki tekstowe (i łańcuchy znaków) działają tak samo jak w wersji
# 2.x

>>> file = open('temp', 'w')                                # Zwraca liczbę zapisanych bajtów
>>> size = file.write('abc\n')                            # Ręczne zamknięcie w celu
>>> file.close()                                         # Domyślnym trybem jest "r" (== "rt"):

yczyszczenia bufora wyjścia

>>> file = open('temp')                                    tekstowe dane wejściowe
>>> text = file.read()                                     # Domyślnym trybem jest "r" (== "rt"):

>>> text
```

```
'abc\n'  
>>> print(text)  
abc
```

Tryby tekstowy i binarny w Pythonie 2.x i 3.x

W Pythonie 2.x nie ma większej różnicy między plikami tekstowymi a binarnymi — oba typy przyjmują i zwracają zawartość w postaciłańcuchów znaków str. Jedyną główną różnicą jest to, że pliki tekstowe automatycznie odwzorowują znaki końca wiersza \n na i z \r\n w systemie Windows, podczas gdy pliki binarne tego nie robią. Poniższe operacje umieszczam tutaj połączone w wierszach z uwagi na zwięzłość takiego zapisu.

```
C:\code> c:\python27\python  
>>> open('temp', 'w').write('abd\n')                                # Zapis w trybie tekstowym –  
dodaje \r  
>>> open('temp', 'r').read()                                         # Wczytanie w trybie  
tekstowym – opuszcza \r  
'abd\n'  
>>> open('temp', 'rb').read()                                         # Wczytanie w trybie binarnym  
– dosłownie  
'abd\r\n'  
>>> open('temp', 'wb').write('abc\n')                                # Zapis w trybie binarnym  
>>> open('temp', 'r').read()                                         # \n nie jest rozszerzane do  
\r\n  
'abc\n'  
>>> open('temp', 'rb').read()  
'abc\n'
```

W Pythonie 3.x wszystko jest nieco bardziej skomplikowane z uwagi na rozróżnienie pomiędzy str (dane tekstowe) i bytes (dane binarne). By to zademonstrować, zapiszemy plik tekstowy i wczytamy go z powrotem w obu trybach wersji 3.x. Warto zauważyc, że dla zapisu musimy przekazać obiekt str, natomiast wczytanie da nam obiekt str lub bytes, w zależności od trybu otwarcia pliku.

```
C:\code> c:\python33\python  
# Zapisanie i wczytanie pliku tekstowego  
>>> open('temp', 'w').write('abc\n')                                # Dane wyjściowe trybu  
tekstowego, przekazanie obiektu str  
4  
>>> open('temp', 'r').read()                                         # Dane wejściowe trybu  
tekstowego, zwrócenie obiektu str  
'abc\n'  
>>> open('temp', 'rb').read()                                         # Dane wejściowe trybu binarnego,  
zwrócenie obiektu bytes
```

```
b'abc\r\n'
```

Warto zwrócić uwagę na to, jak w systemie Windows pliki w trybie tekstowym przekładają znak końca wiersza \n na \r\n dla danych wyjściowych. W przypadku danych wejściowych tryb tekstowy przekłada \r\n z powrotem na \n, jednak tryb binarny tego nie robi. Tak samo jest w wersji 2.x i tak właśnie chcemy. Aby można było przenosić pliki tekstowe do innych systemów operacyjnych, znak końca wiersza musi być zmieniany na \n (typowy dla systemu Linux, w którym znak końca wiersza nie jest zmieniany). Natomiast w przypadku danych binarnych, w których nie istnieje pojęcie końca wiersza, taka konwersja nie może być stosowana. Działanie to można w Pythonie 3.0 kontrolować za pomocą dodatkowych argumentów funkcji open, jeśli jest to pożądane.

Teraz zróbcmy to samo, ale z *plikiem binarnym*. W tym przypadku do zapisania dostarczamy obiekt bytes i nadal otrzymujemy z powrotem obiekt str lub bytes, w zależności od trybu wejścia.

```
# Zapisanie i wczytanie pliku binarnego
>>> open('temp', 'wb').write(b'abc\n')      # Dane wejściowe trybu binarnego,
przekazanie obiektu bytes
4
>>> open('temp', 'r').read()                # Dane wejściowe trybu tekstowego,
zwrócenie obiektu str
'abc\n'
>>> open('temp', 'rb').read()                # Dane wejściowe trybu binarnego,
zwrócenie obiektu bytes
b'abc\n'
```

Warto zwrócić uwagę na to, że znak końca wiersza \n nie jest w przypadku danych wejściowych trybu binarnego rozszerzany do \r\n — znów, jest to pożądany wynik dla danych binarnych. Wymagania w zakresie typów oraz zachowanie pliku są te same, nawet jeśli dane zapisywane do pliku binarnego są z natury prawdziwie binarne. Przykładowo w poniższym kodzie "\x00" to binarny bajt zerowy, który nie jest znakiem wyświetlonym:

```
# Zapisanie i wczytanie prawdziwie binarnych danych
>>> open('temp', 'wb').write(b'a\x00c')    # Przekazanie obiektu bytes
3
>>> open('temp', 'r').read()                # Otrzymanie obiektu str
'a\x00c'
>>> open('temp', 'rb').read()                # Otrzymanie obiektu bytes
b'a\x00c'
```

Pliki w trybie binarnym zawsze zwracają zawartość w postaci obiektu bytes, jednak do zapisu przyjmują obiekty bytes lub bytearray, co jest rozsądne, ponieważ bytearray jest po prostu zmienną odmianą typu bytes. Tak naprawdę większość API w Pythonie 3.x przyjmujących obiekt bytes pozwala także na przekazanie typu bytearray.

```
# Obiekty bytearray także działają
>>> BA = bytearray(b'\x01\x02\x03')
>>> open('temp', 'wb').write(BA)
```

3

```
>>> open('temp', 'r').read()
'\x01\x02\x03'
>>> open('temp', 'rb').read()
b'\x01\x02\x03'
```

Brak dopasowania typu i zawartości w Pythonie 3.x

Warto wiedzieć, że gdy dochodzimy do plików, nie upiecze się nam łamanie reguł związanych z rozróżnieniem typów `str` i `bytes` Pythona. Jak ilustruje to poniższy przykład, jeśli spróbujemy zapisać obiekt `bytes` do pliku tekstowego lub obiekt `str` do pliku binarnego, otrzymamy błędy (skrócone z braku miejsca).

```
# W przypadku zawartości plików typy nie są elastyczne
>>> open('temp', 'w').write('abc\n')          # Tryb tekstowy tworzy i
wymaga obiektu str
4
>>> open('temp', 'w').write(b'abc\n')
TypeError: must be str, not bytes
>>> open('temp', 'wb').write(b'abc\n')        # Tryb binarny tworzy i wymaga
obiektu bytes
4
>>> open('temp', 'wb').write('abc\n')
TypeError: 'str' does not support the buffer interface
```

Ma to sens — tekst z punktu widzenia danych binarnych nie ma znaczenia, dopóki nie zostanie zakodowany. Choć często można się przemieszczać między tymi typami, kodując obiekty `str` i dekodując `bytes`, zgodnie z wcześniejszymi informacjami z tego rozdziału, zazwyczaj chcemy się trzymać *albo* typu `str` dla danych tekstowych, *albo* typu `bytes` dla danych binarnych. Ponieważ zbiory operacji na obiektach `str` i `bytes` w dużej mierze się pokrywają, wybór ten w przypadku większości programów nie będzie wielkim dilematem (kilka świątniczych tego przykładów można znaleźć w omówieniu narzędzi łańcuchów znaków na końcu niniejszego rozdziału).

Poza ograniczeniami związanymi z typem danych w Pythonie 3.x znaczenie ma także *zawartość pliku*. Pliki wyjściowe w trybie tekstowym wymagają dla swojej zawartości obiektu `str`, a nie `bytes`, dlatego nie da się w tej wersji zapisać prawdziwie binarnych danych do pliku w trybie tekstowym. W zależności od reguł dotyczących kodowania bajty spoza domyślnego zbioru znaków mogą czasami być osadzone w normalnym łańcuchu znaków i zawsze mogą zostać zapisane w trybie binarnym. W wersjach wcześniejszych niż 3.3 niektóre z pokazanych niżej operacji wyświetlających ciągi znaków powodują zgłoszenie błędu, natomiast operacje na plikach wykonywane są pomyślnie:

```
# Nie da się wczytać prawdziwie binarnych danych w trybie tekstowym
>>> chr(0xFF)          # FF jest poprawnym znakiem, FE
nie
'ÿ'
```

```

>>> chr(0xFE)                                # W niektórych wersjach Pythona
ta operacja powoduje błąd

>>> open('temp', 'w').write(b'\xFF\xFE\xFD') # Nie można użyć dowolnych
bajtów!

TypeError: must be str, not bytes

>>> open('temp', 'w').write('\xFF\xFE\xFD')   # Można zapisać, jeśli da się
osadzić w obiekcie str

3

>>> open('temp', 'wb').write(b'\xFF\xFE\xFD') # Można także zapisać w trybie
binarnym

3

>>> open('temp', 'rb').read()                 # Zawsze można wczytać jako
binarny obiekt bytes

b'\xff\xfe\xfd'

>>> open('temp', 'r').read()                  # Nie da się wczytać tekstu,
jeśli nie da się go zdekodować!

'ÿ\xfe\xfd'                                    # W niektórych wersjach Pythona
ta operacja powoduje błąd

```

Ponieważ jednak pliki wejściowe w trybie tekstowym w Pythonie 3.x muszą być w stanie zdekodować zawartość zgodnie z kodowaniem Unicode, nie istnieje żaden sposób wczytania prawdziwie binarnych danych w trybie tekstowym, co opisane jest w kolejnym podrozdziale.

Wykorzystywanie plików Unicode

Dotychczas wczytywaliśmy i zapisywaliśmy proste pliki tekstowe oraz binarne. Okazuje się, że odczytanie i zapisanie tekstu Unicode zapisanego w pliku jest proste, ponieważ w Pythonie 3.x wywołanie `open` przyjmuje kodowanie plików tekstowych i w trakcie transferu danych automatycznie je dla nas koduje i dekoduje. Pozwala to na przetwarzanie tekstu Unicode utworzonego z wykorzystaniem schematu kodowania innego od domyślnego dla platformy, a także przechowanie go w innym kodowaniu w celu konwersji.

Odczyt i zapis Unicode w Pythonie 3.x

Tak naprawdę możemyłać znaków przekształcić na inne kodowanie zarówno ręcznie, za pomocą wywołań metod, jak i automatycznie przy wczytaniu z pliku i zapisie do niego. W niniejszym podrozdziale w celach demonstracyjnych wykorzystamy następującyłać Unicode:

```

C:\code> c:\python33\python

>>> S = 'A\xc4B\xe8C'                      # Laćuch pięcioznakowy spoza ASCII
>>> S
'AÄBëC'
>>> len(S)

```

Kodowanie ręczne

Jak już wiemy, taki łańcuch znaków możemy zawsze ręcznie zakodować na surowe bajty zgodnie z docelową nazwą kodowania:

```
# Ręczne kodowanie za pomocą metod
>>> L = S.encode('latin-1')          # 5 bajtów, jeśli zakodowany jako latin-
1
>>> L
b'A\xc4B\xe8C'
>>> len(L)
5
>>> U = S.encode('utf-8')           # 7 bajtów, jeśli zakodowany jako utf-8
>>> U
b'A\xc3\x84B\xc3\xa8C'
>>> len(U)
7
```

Kodowanie danych wyjściowych pliku

By teraz zapisać nasz łańcuch znaków do pliku tekstowego z określonym schematem kodowania, możemy po prostu przekazać pożdaną nazwę kodowania do funkcji open. Choć moglibyśmy najpierw ręcznie zakodować łańcuch, a następnie zapisać go w trybie binarnym, nie musimy tego robić.

```
# Automatycznie kodowanie przy zapisie
>>> open('latindata', 'w', encoding='latin-1').write(S)      # Zapisane jako
Latin-1
5
>>> open('utf8data', 'w', encoding='utf-8').write(S)        # Zapisane jako
UTF-8
5
>>> open('latindata', 'rb').read()                          # Wczytanie
surowych bajtów
b'A\xc4B\xe8C'
>>> open('utf8data', 'rb').read()                            # Różnice w
plikach
b'A\xc3\x84B\xc3\xa8C'
```

Dekodowanie danych wejściowych pliku

W podobny sposób, by wczytać dowolne dane Unicode, przekazujemy po prostu nazwę typu kodowania pliku do funkcji open, która automatycznie dekoduje go z surowych bajtów do łańcuchów znaków. Moglibyśmy również wczytać surowe bajty i ręcznie je zdekodować, ale

przy wczytywaniu w blokach może to być podchwytliwe (możemy wczytać niepełny znak) i na dodatek nie jest niezbędne.

```
# Automatycznie dekodowanie przy odczycie
>>> open('latindata', 'r', encoding='latin-1').read()      # Dekodowane przy
   wczytaniu
'AÄBèC'

>>> open('utf8data', 'r', encoding='utf-8').read()        # Zgodnie z typem
   kodowania
'AÄBèC'

>>> X = open('latindata', 'rb').read()                   # Ręczne
   dekodowanie:
>>> X.decode('latin-1')                                 # Nie jest
   konieczne
'AÄBèC'

>>> X = open('utf8data', 'rb').read()
>>> X.decode()                                         # utf-8 to
   kodowanie domyślne
'AÄBèC'
```

Dekodowanie błędnych dopasowań

Wreszcie należy pamiętać, że takie zachowanie plików w Pythonie 3.x ogranicza rodzaj zawartości, którą możemy załadować jako tekst. Zgodnie z informacjami z poprzedniego podrozdziału Python 3.x musi tak naprawdę być w stanie zdekodować dane z plików tekstowych do łańcucha znaków str zgodnie z nazwą kodowania Unicode — domyślną lub przekazaną. Przykładowo próba otwarcia pliku z prawdziwie binarnymi danymi w trybie tekstowym raczej w wersji 3.x nie zadziała, nawet jeśli użyjemy poprawnego typu obiektu:

```
>>> file = open('C:\Python33\python.exe', 'r')
>>> text = file.read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2: ...
>>> file = open('C:\Python33\python.exe', 'rb')
>>> data = file.read()
>>> data[:20]
b'MZ\x90\x00\x03\x00\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\x00\xb8\x00\x00\x00'
```

Pierwszy z powyższych przykładów mógłby w Pythonie 2.x zadziałać (normalne pliki nie dekodują tekstu), nawet jeśli naprawdę nie powinien. Wczytanie tego pliku może zwrócić w łańcuchu znaków niepoprawne dane z uwagi na automatyczne translacje znaków końca wiersza w trybie tekstowym (wszystkie osadzone bajty \r\n zostaną w systemie Windows przetłumaczone na \n). By potraktować zawartość pliku w Pythonie 2.x jako tekst Unicode, musimy użyć specjalnych narzędzi zamiast ogólnej funkcji wbudowanej open, co zobaczymy za moment. Najpierw jednak zajmijmy się nieco bardziej wybuchowym zagadnieniem...

Obsługa BOM w Pythonie 3.x

Zgodnie z opisem z wcześniejszej części niniejszego rozdziału niektóre schematy kodowania przechowują specjalną sekwencję *znacznika kolejności bajtów* (BOM) na początku plików w celu określenia kolejności *little-* lub *big-endian* danych bądź zadeklarowania typu kodowania. Python pomija ten znacznik przy odczycie i zapisuje go przy zapisie, jeśli sugeruje to nazwa kodowania, jednak czasami musimy użyć specjalnej nazwy kodowania w celu jawnego wymuszenia przetwarzania sekwencji BOM.

Na przykład w standardach UTF-16 i UTF-32 znacznik BOM służy do określania formatu *little-endian* lub *big-endian*. Plik tekstowy zapisany w standardzie UTF-8 też może zawierać ten znacznik, jednak nie jest on wymagany i oznacza tylko tyle, że plik jest zapisany w powyższym standardzie. Podczas odczytywania i zapisywania danych znacznik jest lub nie jest uwzględniany — w zależności od domyślnego lub jawnie wskazanego standardu kodowania, tj.:

- Jeżeli stosowany jest standard UTF-16, wtedy znacznik BOM jest zawsze stosowany w plikach zapisanych w tym standardzie. Można stosować bardziej precyzyjną nazwę "utf-16-le" reprezentującą format *little-endian*.
- Jeżeli stosowany jest standard UTF-8, wtedy użycie nazwy "utf-8-sig" powoduje, że znacznik BOM jest pomijany podczas odczytywania pliku i uwzględniany podczas jego zapisywania, co nie ma miejsca w przypadku użycia nazwy "utf-8".

Pomijanie znacznika BOM w Notatniku

Utwórzmy teraz kilka plików, aby przekonać się, jak wygląda stosowanie znacznika BOM w praktyce. Jeśli zapiszemy plik tekstowy w Notatniku systemu Windows, możemy określić jego typ kodowania zgodnie z listą rozwijaną — zwykły tekst ASCII, UTF-8 lub UTF-16 w wersji *little-* lub *big-endian*. Jeśli jednowierszowy plik tekstowy o nazwie *spam.txt* zapiszemy w Notatniku jako typ kodowania *ANSI*, zostanie on zapisany jako prosty tekst ASCII bez znacznika BOM. Kiedy plik ten zostanie wczytany w Pythonie w trybie binarnym, zobaczymy prawdziwe bajty w nim przechowane. Po wczytaniu jako tekst Python domyślnie wykonuje translacje końca wierszy. Możemy zdekodować plik jako tekst UTF-8, ponieważ ASCII jest podzbiorem tego schematu (a UTF-8 jest kodowaniem domyślnym Pythona 3.x).

```
c:\code> C:\Python33\python                                # Plik zapisany w Notatniku
>>> import sys, locale
>>> locale.getpreferredencoding(False)
'cp1252'
>>> open('spam.txt', 'rb').read()                         # Plik tekstowy ASCII (utf-8)
b'mielonka\r\nMIELONKA\r\n'
>>> open('spam.txt', 'r').read()                           # Tryb tekstowy tłumaczy znaki
końca wiersza
'mielonka\nMIELONKA\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'mielonka\nMIELONKA\n'
```

Jeśli zamiast tego plik ten zapiszemy w Notatniku jako *UTF-8*, zostanie on poprzedzony trzybajtową sekwencją BOM dla UTF-8 i w celu pominięcia znacznika przez Pythona musimy podać bardziej uszczegółowioną nazwę kodowania („utf-8-sig”).

```
>>> open('spam.txt', 'rb').read()                         # UTF-8 z 3-bajtowym znacznikiem
BOM
b'\xef\xbb\xbfmielonka\r\nMIELONKA'
```

```

>>> open('spam.txt', 'r').read()
'd»żmielonka\nMIELONKA\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'\ufeффmielonka\nMIELONKA\n'
>>> open('spam.txt', 'r', encoding='utf-8-sig').read()
'mielonka\nMIELONKA\n'

```

Jeśli plik przechowamy w Notatniku jako „Unicode big-endian”, otrzymamy w pliku dane w formacie UTF-16, poprzedzone dwubajtową sekwencją BOM. Nazwa kodowania „utf-16” w Pythonie pomija BOM, ponieważ sekwencja ta jest znana (gdyż wszystkie pliki UTF-16 zawierają BOM), natomiast nazwa „utf-16-be” obsługuje format big-endian, ale nie pomija znacznika BOM (drugie polecenie w poniższym przykładzie użyte w starszej wersji Pythona zakończy się niepowodzeniem):

```

>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00m\x00i\x00e\x00l\x00o\x00n\x00k\x00a\x00\r\x00\n\x00M\x00I\x00E
\x00L\x00O\x00N\x00K\x00A'

>>> open('spam.txt', 'r').read()
'\xe7\xfe\x00m\x00i\x00e\x00l\x00o\x00n\x00k\x00a\x00\r\x00\n\x00M\x00I\x00E\x00
\x00L\x00O\x00N\x00K\x00A\x00\r\x00\n\x00'

>>> open('spam.txt', 'r', encoding='utf-16').read()
'mielonka\nMIELONKA\n'

>>> open('spam.txt', 'r', encoding='utf-16-be').read()
'\ufeффmielonka\nMIELONKA'

```

Nawiasem mówiąc, kodowanie Unicode stosowane w Notatniku jest w rzeczywistości formatem UTF-16 little-endian, który oczywiście jest jednym z wielu wariantów powyższego rodzaju kodowania.

Pomijanie znacznika BOM w Pythonie

Tak samo będzie zazwyczaj w przypadku *danych wyjściowych*. Przy zapisywaniu pliku Unicode w kodzie Pythona potrzebna nam bardziej uszczegółowiona nazwa kodowania, by wymusić w UTF-8 znacznik BOM. Zwykle „utf-8” nie zapisuje znacznika BOM (lub go pomija), natomiast „utf-8-sig” to robi:

```

>>> open('temp.txt', 'w', encoding='utf-8').write('mielonka\nMIELONKA\n')
18
>>> open('temp.txt', 'rb').read()                                     # Brak BOM
b'mielonka\r\nMIELONKA\r\n'
>>> open('temp.txt', 'w', encoding='utf-8-sig').write('mielonka\nMIELONKA\n')
18
>>> open('temp.txt', 'rb').read()                                       # Zapisanie BOM
b'\xef\xbb\xbfmielonka\r\nMIELONKA\r\n'

```

```

>>> open('temp.txt', 'r').read()
'đ»żmielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()          # Zachowanie BOM
'\ufeffmielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()      # Pominięcie BOM
'mielonka\nMIELONKA\n'

```

Warto zauważyc, że choć „utf-8” nie pomija BOM, dane *bez* BOM można wczytać zarówno za pomocą nazwy kodowania „utf-8”, jak i „utf-8-sig”. To drugie rozwiązanie należy zastosować w przypadku danych wejściowych, jeśli nie jesteśmy pewni, czy sekwencja BOM obecna jest w pliku.

```

>>> open('temp.txt', 'w').write('mielonka\nMIELONKA\n')
18
>>> open('temp.txt', 'rb').read()                         # Dane bez BOM
b'mielonka\r\nMIELONKA\r\n'
>>> open('temp.txt', 'r').read()                          # Działa dowolna nazwa
kodowania UTF-8
'mielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()
'mielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'mielonka\nMIELONKA\n'

```

Wreszcie w przypadku nazwy kodowania „utf-16” sekwencja BOM obsługiwana jest automatycznie. W przypadku *danych wyjściowych* dane te zapisywane są zgodnie z ustawieniem kolejności bajtów platformy, a znacznik BOM jest zawsze zapisywany. Dla *danych wejściowych* są one dekodowane zgodnie z sekwencją BOM, a sekwencja ta jest zawsze wycinana, ponieważ jest to cecha tego schematu:

```

>>> sys.byteorder
'little'
>>> open('temp.txt', 'w', encoding='utf-16').write('mielonka\nMIELONKA\n')
18
>>> open('temp.txt', 'rb').read()
b'\xff\xfe\x00i\x00e\x00l\x00o\x00n\x00k\x00a\x00\r\x00\n\x00M\x00I\x00E\x00
L\x00O\x00N\x00K\x00A\x00\r\x00\n\x00'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'mielonka\nMIELONKA\n'

```

Bardziej uszczegółowione nazwy kodowania UTF-16 mogą określać różne ustawienia kolejności bajtów, choć w niektórych sytuacjach możemy być zmuszeni ręcznie zapisywać lub pomijać BOM, jeśli znacznik ten jest wymagany bądź obecny. Poniższy przykład ilustruje bardziej precyzyjne manipulowanie znacznikiem BOM:

```

>>> open('temp.txt', 'w', encoding='utf-16-be').write('\ufeффmielonka\nMIELONKA\n')
19
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00m\x00i\x00e\x00l\x00o\x00n\x00k\x00a\x00\r\x00\n\x00M\x00I\x00E
\x00L\x00O\x00N\x00K\x00A'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'mielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-16-be').read()
'\ufeффmielonka\nMIELONKA\n'

```

Bardziej uszczegółowione nazwy kodowania UTF-16 działają dobrze w przypadku plików bez BOM, jednak nazwa kodowania „utf-16” wymaga tej sekwencji w celu ustalenia kolejności bajtów:

```

>>> open('temp.txt', 'w', encoding='utf-16-le').write('MIELONKA')
8
>>> open('temp.txt', 'rb').read()                      # OK, jeśli BOM nie ma lub nie
oczekiwano
b'M\x00I\x00E\x00L\x00O\x00N\x00K\x00A\x00'
>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'MIELONKA'
>>> open('temp.txt', 'r', encoding='utf-16').read()
UnicodeError: UTF-16 stream does not start with BOM

```

Warto poeksperymentować samodzielnie z tymi typami kodowania lub sprawdzić dokumentację biblioteki Pythona pod kątem szczegółowych informacji o znaczniku kolejności bajtów BOM.

Pliki Unicode w Pythonie 2.x

Powyższe omówienie ma zastosowanie do typów łańcuchów znaków oraz plików z Pythona 3.x. Podobny efekt można uzyskać dla plików Unicode w wersji 2.x, jednak interfejs jest nieco odmienny. Jeśli zastąpimy typ `str` za pomocą `unicode`, a funkcję `open` funkcją `codecs.open`, wynik będzie w wersji 3.x mniej więcej taki sam.

```

C:\code> c:\python27\python
>>> S = u'A\xc4B\xe8C'                      # Typ w wersji 2.x
>>> print S
AÄBèC
>>> len(S)
5
>>> S.encode('latin-1')                      # Jawne określenie kodowania
'A\xc4B\xe8C'

```

```

>>> S.encode('utf-8')
'A\xc3\x84B\xc3\xa8C'
>>> import codecs # Pliki w wersji 2.x
>>> codecs.open('latindata', 'w', encoding='latin-1').write(S)
# Zapisanie zakodowanego tekstu
>>> codecs.open('utfdata', 'w', encoding='utf-8').write(S)
>>> open('latindata', 'rb').read()
'A\xc4B\xe8C'
>>> open('utfdata', 'rb').read()
'A\xc3\x84B\xc3\xa8C'
>>> codecs.open('latindata', 'r', encoding='latin-1').read()
# Odczytanie zakodowanego tekstu
u'A\xc4B\xe8C'
>>> codecs.open('utfdata', 'r', encoding='utf-8').read()
u'A\xc4B\xe8C'
>>> print codecs.open('utfdata', 'r', encoding='utf-8').read()
# Wyświetlenie odczytanych danych
AÃBèC

```

Więcej szczegółowych informacji o kodowaniu Unicode w wersji 2.x jest zawartych w poprzednich podrozdziałach i dokumentacji do tej wersji Pythona.

Unicode w nazwach plików i w strumieniach

W tej części rozdziału skupiliśmy się głównie na standardzie Unicode kodowania treści plików tekstowych. Jednak w Pythonie znaki spoza zakresu ASCII można stosować również w *nazwach* plików. Sposób kodowania określają niezależne opcje w module sys, które mogą różnić się w zależności od wersji Pythona i systemu operacyjnego. Poniższy kod uruchomiony w wersji 2.x w systemie Windows zwróci na pierwszym miejscu nazwę 'ascii'.

```

>>> import sys
>>> sys.getdefaultencoding(), sys.getfilesystemencoding() # Kodowanie
treści i nazw plików
('utf-8', 'mbcs')

```

Nazwy plików: znaki i bajty

Kodowanie nazw plików jest zazwyczaj kwestią drugorzędną. W skrócie: jeżeli w argumencie metody open umieści się ciąg Unicode, wtedy zostanie on automatycznie zakodowany zgodnie z ustawieniami systemu operacyjnego. Jeżeli natomiast użyje się dowolnego ciągu bajtów (dotyczy to nie tylko argumentu metody open, ale też metod zmieniających i wyświetlających zawartości katalogów), wtedy domyślne kodowanie jest ignorowane i nazwa pliku jest pozostawiana w niezakodowanej formie. Jest to użyteczna konwencja w przypadku, gdy nazwa pliku zawiera znaki charakterystyczne dla danego systemu operacyjnego, jak w poniższym

przykładzie (używam systemu Windows, więc niektóre z poniższych poleceń wykonane w innym systemie mogą zakończyć się błędem):

```
>>> f = open('xxx\u00A5', 'w')          # Nazwa pliku zawierająca znak spoza ASCII  
>>> f.write('\xA5999\n')             # Zapisanie w pliku pięciu znaków  
>>> f.close()  
>>> print(open('xxx\u00A5').read())    # Nazwa tekstowa, automatycznie kodowana  
¥999  
>>> print(open(b'xxx\xA5').read())    # Nazwa bajtowa, już zakodowana  
¥999  
>>> import glob                      # Moduł do rozszerzania nazw plików  
>>> glob.glob('*\u00A5*')            # Dekodowanie tekstu  
['xxx¥']  
>>> glob.glob(b'*\xA5*')            # Kodowanie bajtów  
[b'xxx\xA5']
```

Zawartość strumienia: zmienna PYTHONIOENCODING

Do określenia rodzaju kodowania treści wysyłanych do standardowych strumieni (wejścia, wyjścia i błędów) można użyć zmiennej środowiskowej PYTHONIOENCODING. Jej wartość zastępuje domyślne ustawienie kodowania wyświetlanego tekstu, tj. systemowe Windows w wersji 3.x lub ASCII w wersji 2.x. Ustawienie za pomocą tej zmiennej ogólnego kodowania Unicode, na przykład UTF-8, może być niezbędne do poprawnego wyświetlania w konsoli znaków spoza zestawu ASCII (dodatekowo w systemie Windows może być wymagana zmiana strony kodowej). Na przykład skrypt wyświetlający nazwy plików zawierające znaki inne niż ASCII może nie działać poprawnie, jeżeli nie użyje się tej zmiennej.

Więcej informacji na ten temat zawartych jest w rozdziale 25., w punkcie „Symbole walut: Unicode w akcji”. Przedstawiony tam przykład demonstruje istotę kodowania Unicode oraz znaczenie zmiennej PYTHONIOENCODING i związane z jej użyciem wymagania. Nie będziemy tych zagadnień tutaj ponownie analizować.

Szerzej na temat kodowania w ogólności traktują książki o Pythonie, na przykład *Programming Python* (wydanie 4 lub nowsze). W tej książce dokładnie opisane są strumienie i pliki pod kątem ich użycia w aplikacjach.

Inne zmiany w narzędziach do przetwarzania łańcuchów znaków w Pythonie 3.x

Niektóre z popularnych narzędzi służących w bibliotece standardowej Pythona do przetwarzania łańcuchów znaków zostały zmodyfikowane pod kątem dychotomii nowych typów str i bytes. Nie będziemy omawiać tych skoncentrowanych na aplikacjach narzędzi zbyt szczegółowo w książce poświęconej jądru języka, jednak na zakończenie niniejszego rozdziału

przedstawię krótki przegląd czterech najważniejszych narzędzi, na które zmiana ta miała wpływ: modułu dopasowywania wzorców `re`, modułu danych binarnych `struct`, modułu serializacji obiektów `pickle` oraz pakietu `xml` służącego do analizy składniowej tekstu XML. Jak już wspomniałem, inne narzędzia, na przykład moduł `json`, uległy podobnym do opisanych tutaj zmianom.

Moduł dopasowywania wzorców `re`

Moduł dopasowywania wzorców Pythona `re` obsługuje przetwarzanie tekstu w sposób bardziej ogólny niż za pomocą prostych metod łańcuchów znaków, takich jak `find`, `split` oraz `replace`. Dzięki `re` łańcuchy znaków wyznaczające cele wyszukiwania i dzielenia można opisać za pomocą ogólnych wzorców, a nie tekstu. Moduł ten został uogólniony w taki sposób, by działać na obiektach dowolnego typu łańcucha znaków z Pythona 3.x — `str`, `bytes` oraz `bytearray` — i zwracać wynikowe podłańcuchy znaków tego samego typu co podmiot operacji. W wersji 2.x moduł `re` obsługuje oba łańcuchy: `unicode` i `str`.

Oto przykład działania tego modułu w wersji 3.x, dokonujący ekstrakcji podłańcuchów znaków z wiersza tekstu, zaczerpniętego oczywiście z *Sensu życia* Monty Pythona. W przypadku łańcuchów wzorców `(.*)` oznacza dowolny znak `(.)`, zero lub więcej razy `(*)`, zapisany jako dopasowany podłańcuch znaków `(())`. Części łańcucha znaków dopasowane za pomocą wzorca zawartego w nawiasach dostępne są po dopasowaniu za pomocą metod `group` lub `groups`.

```
C:\code> c:\python33\python
>>> import re
>>> S = 'Dzielny sir Robin nawiewa raz po raz'                      # Wiersz tekstu
>>> B = b'Dzielny sir Robin nawiewa raz po raz'                      # Zazwyczaj z
pliku
>>> re.match('(.*) sir (.*) nawiewa (.*)', S).groups()                # Dopasowanie
wiersza do wzorca
('Dzielny', 'Robin', 'raz po raz')                                       # Dopasowane
podłańcuchy znaków
>>> re.match(b'(.*) sir (.*) nawiewa (.*)', B).groups()                # Podłańcuchy
znaków bytes
(b'Dzielny', b'Robin', b'raz po raz')
```

W Pythonie 2.x wyniki są podobne, jednak dla tekstu spoza zakresu ASCII wykorzystywany jest typ `unicode`, a typ `str` obsługuje zarówno tekst 8-bitowy, jak i binarny:

```
C:\code> c:\python27\python
>>> import re
>>> S = 'Dzielny sir Robin nawiewa raz po raz'                      # Prosty tekst
>>> U = u'Dzielny sir Robin nawiewa raz po raz'                      # Binarny tekst
Unicode
>>> re.match('(.*) sir (.*) nawiewa (.*)', S).groups()
('Dzielny', 'Robin', 'raz po raz')
>>> re.match('(.*) sir (.*) nawiewa (.*)', U).groups()
(u'Dzielny', u'Robin', u'raz po raz')
```

Ponieważ typy `bytes` i `str` obsługują w zasadzie ten sam zbiór operacji, rozróżnienie pomiędzy nimi jest praktycznie niezauważalne. Warto jednak zauważyć, że tak jak w innych API, nie można mieszać typów `str` i `bytes` w argumentach ich wywołań w Pythonie 3.x (choć jeśli nie planujemy dopasowywać wzorców na danych binarnych, najprawdopodobniej nie musimy się tym przejmować).

```
C:\code> c:\python33\python
>>> import re
>>> S = 'Dzielny sir Robin nawiewa raz po raz'
>>> B = b'Dzielny sir Robin nawiewa raz po raz'
>>> re.match('(.*) sir (.*) nawiewa (.*)', B).groups()
TypeError: can't use a string pattern on a bytes-like object
>>> re.match(b'(.*) sir (.*) nawiewa (.*)', S).groups()
TypeError: can't use a bytes pattern on a string-like object
>>> re.match(b'(.*) sir (.*) nawiewa (.*)', bytearray(B)).groups()
(bytearray(b'Dzielny'), bytearray(b'Robin'), bytearray(b'raz po raz'))
>>> re.match('(.*) sir (.*) nawiewa (.*)', bytearray(B)).groups()
TypeError: can't use a string pattern on a bytes-like object
```

Moduł danych binarnych `struct`

Moduł `struct` Pythona, wykorzystywany do tworzenia i ekstrakcji spakowanych danych binarnych z łańcuchów znaków, także działa w Pythonie 3.x tak samo jak w 2.x, jednak spakowane dane reprezentowane są jedynie w postaci obiektów `bytes` i `bytearray`, a nie typu `str` (co ma sens, biorąc pod uwagę, że przeznaczony jest on do przetwarzania danych binarnych, a nie dowolnie zakodowanego tekstu). Począwszy od wersji 3.2, kod "s" reprezentuje ciąg typu `bytes` (automatyczne kodowanie ciągu typu `str`, stosowane w starszych wersjach, nie jest już wykonywane).

Poniżej znajduje się przykład działania obu wersji Pythona, pakujący trzy obiekty do łańcucha znaków zgodnie ze specyfikacją typu binarnego (tworzą one czterobajtową liczbę całkowitą, czterobajtowy łańcuch znaków oraz dwubajtową liczbę całkowitą).

```
C:\code> c:\python33\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'jajo', 8)           # Typ bytes w wersji 3.x (8-bitowy
                                             # łańcuch znaków)
b'\x00\x00\x00\x07jajo\x00\x08'

C:\code> c:\python27\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'jajo', 8)           # Typ str w wersji 2.x (8-bitowy
                                             # łańcuch znaków)
'\x00\x00\x00\x07jajo\x00\x08'
```

Ponieważ obiekt `bytes` ma interfejs prawie identyczny jak `str` w wersjach 3.x oraz 2.x, większość programistów nie będzie najprawdopodobniej musiała sobie tym zawracać głowy — zmiany są dla większości istniejącego kodu nieistotne, zwłaszcza że wczytanie z pliku binarnego automatycznie tworzy obiekt `bytes`. Choć ostatni test z poniższego przykładu nie działa przy braku dopasowania typów, większość skryptów wczyta dane binarne z pliku, a nie utworzy je w postaci łańcucha znaków.

```
C:\code> c:\python33\python
>>> import struct
>>> B = struct.pack('>i4sh', 7, 'jajo', 8)
>>> B
b'\x00\x00\x00\x07jajo\x00\x08'
>>> vals = struct.unpack('>i4sh', B)
>>> vals
(7, b'jajo', 8)
>>> vals = struct.unpack('>i4sh', B.decode())
TypeError: 'str' does not have the buffer interface
```

Poza nową składnią obiektu `bytes` tworzenie i wczytywanie plików binarnych działa w wersji 3.x prawie tak samo jak w Pythonie 2.x. Kod taki jak poniższy jest jednym z głównych miejsc, w których programiści zauważają typ obiektu `bytes`.

```
C:\code> c:\python33\python
# Zapisanie wartości do spakowanego pliku binarnego
>>> F = open('data.bin', 'wb')                      # Otwarcie binarnego
pliku wyjścia
>>> import struct
>>> data = struct.pack('>i4sh', 7, 'jajo', 8)        # Utworzenie spakowanych
danych binarnych
>>> data                                         # W wersji 3.x typ bytes
3.0, nie str
b'\x00\x00\x00\x07jajo\x00\x08'
>>> F.write(data)                                # Zapisanie do pliku
10
>>> F.close()
# Wczytanie wartości ze spakowanego pliku binarnego
>>> F = open('data.bin', 'rb')                      # Otwarcie binarnego
pliku wejścia
>>> data = F.read()                               # Wczytanie obiektu
bytes
>>> data
b'\x00\x00\x00\x07jajo\x00\x08'
```

```

>>> values = struct.unpack('>i4sh', data)          # Ekstrakcja spakowanych
danych binarnych

>>> values                                       # Powrót do obiektów
Pythona

(7, b'jajo', 8)

```

Po ekstrakcji spakowanych danych binarnych do obiektów Pythona, jak wyżej, jeśli musimy, możemy jeszcze bardziej zagłębić się w świat binarny — łańcuchy znaków można indeksować czy tworzyć z nich wycinki w celu pobrania wartości poszczególnych bajtów. Poszczególne bity można z kolei pobierać z liczb całkowitych za pomocą operatorów poziomu bitowego (więcej informacji na temat zastosowanych tutaj działań można znaleźć we wcześniejszej części książki).

```

>>> values                                         # Wynik struct.unpack

(7, b'jajo', 8)

# Dostęp do bitów przeanalizowanych składowo liczb całkowitych

>>> bin(values[0])                                # Może dojść do bitów w liczbach
całkowitych

'0b111'

>>> values[0] & 0x01                               # Sprawdzenie pierwszego
(najniższego) bitu w liczbie całkowitej

1

>>> values[0] | 0b1010                            # Bitowe or: załączenie bitów

15

>>> bin(values[0] | 0b1010)                      # Dziesiętne 15 to binarne 1111

'0b1111'

>>> bin(values[0] ^ 0b1010)                      # Bitowe xor: wyłączenie, jeśli oba
są prawdziwe

'0b1101'

>>> bool(values[0] & 0b100)                      # Sprawdzenie, czy bit 3 jest
włączony

True

>>> bool(values[0] & 0b1000)                      # Sprawdzenie, czy bit 4 jest
ustawiony

False

```

Ponieważ łańcuchy znaków bytes po analizie składniowej są sekwencjami niewielkich liczb całkowitych, w podobny sposób możemy przetwarzać ich poszczególne bajty:

```

# Dostęp do bajtów łańcuchów znaków po analizie składniowej, a także do
znajdujących się w nich bitów

>>> values[1]

b'jajo'

```

```

>>> values[1][0]                                # Łańcuch znaków bytes: sekwencja
liczba całkowitych

106

>>> values[1][1:]                                # Wyświetlane jako znaki ASCII
b'ajo'

>>> bin(values[1][0])                            # Może dostać się do bitów bajtów
z łańcuchów znaków

'0b1101010'

>>> bin(values[1][0] | 0b1100)                  # Załączenie bitów

'0b1101110'

>>> values[1][0] | 0b1100

110

```

Oczywiście większość programistów Pythona nie zajmuje się bitami binarnymi. Python ma typy obiektów wyższego poziomu, takie jak listy i słowniki, które są najczęściej lepszą metodą reprezentowania informacji w skryptach napisanych w tym języku. Jeśli jednak musimy wykorzystywać bądź przetwarzać dane niskopoziomowe używane w programach w języku C, bibliotekach sieciowych czy innych interfejsach, Python udostępnia narzędzia mogące nas w tym wspomóc.

Moduł serializacji obiektów pickle

Z modułem pickle spotkaliśmy się krótko w rozdziałach 9, 29. i 31. W rozdziale 28. używaliśmy także modułu `shelve`, który wewnętrznie wykorzystuje moduł `pickle`. Dla pełności obrazu pamiętajmy, że wersja modułu `pickle` z Pythona 3.x zawsze tworzy obiekt `bytes`, bez względu na domyślny bądź przekazany „protokół” (poziom formatu danych). Możemy się o tym przekonać, wykorzystując wywołanie `dumps` z tego modułu w celu zwrócenia łańcucha znaków `pickle` dla obiektu.

```

C:\code> C:\Python33\python

>>> import pickle                                # dumps() zwraca łańcuch znaków
pickle

>>> pickle.dumps([1, 2, 3])                      # Domyślny protokół Pythona 3.x:
protocol=3, czyli binarny
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'

>>> pickle.dumps([1, 2, 3], protocol=0)          # Protokół ASCII (0), ale nadal
obiekt bytes!

b'(lp0\nL1L\naL2L\naL3L\na.'

```

Wynika z tego, że pliki wykorzystywane do przechowywania zserializowanych za pomocą modułu `pickle` obiektów w Pythonie 3.x muszą zawsze być otwierane w *trybie binarnym*, ponieważ pliki tekstowe wykorzystują do reprezentowania danych łańcuchy znaków `str`, a nie `bytes`. Wywołanie `dumps` próbuje po prostu zapisać zserializowany łańcuch do otwartego pliku wyjściowego.

```

>>> pickle.dump([1, 2, 3], open('temp', 'w'))      # Pliki tekstowe nie
działają na obiektach bytes!

```

```

TypeError: can't write bytes to text stream      # Pomimo wartości protokołu
>>> pickle.dump([1, 2, 3], open('temp', 'w'), protocol=0)
TypeError: can't write bytes to text stream
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))    # W wersji 3.x zawsze
wykorzystuje tryb binarny
>>> open('temp', 'r').read()                      # Ta instrukcja działa
poprawnie, ale przez przypadek
'\u20ac\x03]q\x00(K\x01K\x02K\x03e.'

```

Należy zwrócić uwagę, że ostatnia instrukcja użyta w powyższym przykładzie, otwierająca plik w trybie tekstowym, nie spowodowała błędu tylko dlatego, że zawarte w pliku dane binarne są zgodne z domyślnym dekoderem UTF-8 stosowanym w systemie Windows. Jest to jednak czysty przypadek. Co więcej, instrukcja ta, użyta w starszej wersji Pythona lub w innym systemie operacyjnym, zakończyłaby się błędem.

Ponieważ zserializowane dane nie są tekstem Unicode, który można zdekodować, tak samo jest w przypadku danych wejściowych — poprawne użycie w Pythonie 3.x wymaga zawsze zapisywania i wczytywania zserializowanych danych w trybach binarnych:

```

>>> pickle.dump([1, 2, 3], open('temp', 'wb'))
>>> pickle.load(open('temp', 'rb'))
[1, 2, 3]
>>> open('temp', 'rb').read()
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'

```

W Pythonie 2.x (oraz wcześniejszych wersjach) możemy dla zserializowanych danych wykorzystywać także pliki w trybie tekstowym, o ile protokół ustawiony jest na poziom 0 (ustawienie domyślne w Pythonie 2.6) i w spójny sposób wykorzystujemy tryb tekstowy do przekształcania końca wierszy.

```

C:\code> c:\python27\python
>>> import pickle
>>> pickle.dumps([1, 2, 3])                         # W Pythonie 2.x domyślny
protokół to 0, czyli ASCII
'(lp0\nI1\naI2\naI3\na.'
>>> pickle.dumps([1, 2, 3], protocol=1)
']q\x00(K\x01K\x02K\x03e.'
>>> pickle.dump([1, 2, 3], open('temp', 'w'))    # W Pythonie 2.x działa tryb
tekstowy
>>> pickle.load(open('temp'))
[1, 2, 3]
>>> open('temp').read()
'(lp0\nI1\naI2\naI3\na.'

```

Jeśli ma dla nas znaczenie neutralność względem wersji Pythona albo nie chcemy się przejmować protokołami czy innymi wartościami domyślnymi określonej wersji, należy w

przypadku zserializowanych danych zawsze używać plików w trybie binarnym. Poniższy kod działa tak samo w Pythonie 3.x oraz 2.x.

```
>>> import pickle  
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))      # Neutralne względem wersji  
>>> pickle.load(open('temp', 'rb'))                # I wymagane w Pythonie 3.x  
[1, 2, 3]
```

Ponieważ prawie wszystkie programy pozwalają Pythonowi na automatyczną serializację oraz deserializację obiektów i nie zajmują się zawartością samych zserializowanych danych, wymaganie użycia binarnego trybu plików jest jedyną istotną niezgodnością nowego modelu serializacji Pythona 3.x z poprzednimi wersjami. Więcej informacji na temat serializacji obiektów można znaleźć w innych publikacjach książkowych oraz dokumentacji Pythona.

Narzędzia do analizy składniowej XML

XML to język znaczników służący do definiowania ustrukturyzowanych informacji, wykorzystywany powszechnie do definiowania dokumentów oraz danych w internecie. Choć część informacji można pobierać z tekstu XML za pomocą podstawowych metod łańcuchów znaków lub modułu dopasowywania wzorców `re`, zagnieźdzanie konstrukcji XML i dowolne nazwy atrybutów umożliwiają dokładniejsze przetwarzanie dokumentów w tym języku.

Ponieważ XML jest tak wszechobecnym formatem, sam Python zawiera pełny pakiet narzędzi do analizy składniowej XML obsługujący modele SAX oraz DOM, a także pakiet znany jako `ElementTree` („drzewo elementów”) — specyficzne dla Pythona API służące do analizy składniowej oraz konstruowania XML. Poza podstawową analizą składniową obsługę dodatkowych narzędzi XML, takich jak XPath, Xquery i XSLT, można znaleźć w domenie open source.

XML z definicji reprezentuje tekst w formacie Unicode w celu obsługi interakcjonalizacji. Choć większość narzędzi do przetwarzania XML w Pythonie zawsze zwracała łańcuchy znaków Unicode, w Pythonie 3.x ich wyniki zmieniły się z typu `unicode` Pythona 2.x na ogólny typ łańcuchów znaków `str` wersji 3.x. Ma to sens, biorąc pod uwagę, że łańcuch znaków `str` w Pythonie 3.x jest w formacie Unicode, bez względu na to, czy kodowanie to ASCII, czy jakieś inne.

Nie mamy tutaj możliwości za bardzo zagłębić się w szczegóły, ale by nieco zasmakować tej dziedziny wykorzystania Pythona, założymy, że mamy prosty tekstowy plik XML o nazwie `mybooks.xml`:

```
<books>  
    <date>1995~2013</date>  
    <title>Learning Python</title>  
    <title>Programming Python</title>  
    <title>Python Pocket Reference</title>  
    <publisher>O'Reilly Media</publisher>  
</books>
```

i chcemy wywołać skrypt w celu pobrania i wyświetlenia zawartości wszystkich osadzonych znaczników `title`, jak poniżej:

```
Learning Python  
Programming Python
```

Python Pocket Reference

Istnieją przynajmniej cztery proste sposoby dokonania tego (nie licząc narzędzi bardziej zaawansowanych, jak XPath). Po pierwsze, moglibyśmy wykonać podstawowe *dopasowanie wzorców* na tekście pliku, choć jeśli tekst jest nieprzewidywalny, może to być niedokładne. Tam, gdzie ma to zastosowanie, można tego dokonać za pomocą omówionego wcześniej modułu `re` — jego metoda `match` szuka dopasowania na początku łańcucha znaków, `search` szuka dopasowania dalej, a wykorzystana tutaj metoda `findall` lokalizuje wszystkie miejsca, w których wzorzec dopasowany zostaje w łańcuchu znaków (wynik zwracany jest w postaci listy dopasowanych podłańcuchów znaków odpowiadających grupom wzorców w nawiasach lub w postaci krotki takich list w przypadku większej liczby grup).

```
# Plik patternparse.py
import re
text = open('mybooks.xml').read()
found = re.findall('<title>(.*)</title>', text)
for title in found: print(title)
```

Po drugie, by mieć większe możliwości, moglibyśmy wykonać pełną analizę składniową XML za pomocą obsługi *analizy składniowej drzewa DOM* z biblioteki standardowej Pythona. DOM przetwarza tekst XML w drzewo obiektów i udostępnia interfejs do nawigowania w drzewie w celu ekstrakcji atrybutów i wartości znaczników. Interfejs ten jest oficjalną i niezależną od Pythona specyfikacją.

```
# Plik domparse.py
from xml.dom.minidom import parse, Node
xmltree = parse('mybooks.xml')
for node1 in xmltree.getElementsByTagName('title'):
    for node2 in node1.childNodes:
        if node2.nodeType == Node.TEXT_NODE:
            print(node2.data)
```

Trzecią opcją jest obsługa *analizy składniowej SAX* dla XML w bibliotece standardowej Pythona. W modelu SAX metody klas otrzymują wywołania zwrotne w miarę postępu analizy i wykorzystują informacje o stanie do zapamiętywania, w którym miejscu dokumentu się znajdują, oraz zbierania danych.

```
# Plik saxparse.py
import xml.sax.handler
class BookHandler(xml.sax.handler.ContentHandler):
    def __init__(self):
        self.inTitle = False
    def startElement(self, name, attributes):
        if name == 'title':
            self.inTitle = True
    def characters(self, data):
```

```

        if self.inTitle:
            print(data)
    def endElement(self, name):
        if name == 'title':
            self.inTitle = False
import xml.sax
parser = xml.sax.make_parser()
handler = BookHandler()
parser.setContentHandler(handler)
parser.parse('mybooks.xml')

```

Wreszcie system *ElementTree* dostępny w pakiecie *etree* biblioteki standardowej pozwala uzyskać te same rezultaty co narzędzia do analizy składniowej DOM, jednak z wykorzystaniem mniejszej ilości kodu. To specyficzny dla Pythona sposób analizy składniowej oraz generowania tekstu XML. Po przetworzeniu tekstu API tego systemu daje dostęp do komponentów dokumentu.

```

# Plik etreeparse.py
from xml.etree.ElementTree import parse
tree = parse('mybooks.xml')
for E in tree.findall('title'):
    print(E.text)

```

Po wykonaniu w Pythonie 2.x lub 3.x wszystkie cztery skrypty wyświetlają ten sam wynik:

```

C:\code> c:\python27\python domparse.py
Learning Python
Programming Python
Python Pocket Reference
C:\code> c:\python33\python domparse.py
Learning Python
Programming Python
Python Pocket Reference

```

W Pythonie 2.x część z powyższych skryptów zwraca obiekty łańcuchów znaków *unicode*, podczas gdy w wersji 3.x wszystkie zwracają łańcuchy znaków *str*, ponieważ typ ten zawiera tekst Unicode (ASCII bądź inny).

```

C:\code> c:\python33\python
>>> from xml.dom.minidom import parse, Node
>>> xmmtree = parse('mybooks.xml')
>>> for node in xmmtree.getElementsByTagName('title'):
...     for node2 in node.childNodes:

```

```

...
    if node2.nodeType == Node.TEXT_NODE:
        node2.data
...
'Learning Python'
'Programming Python'
'Python Pocket Reference'
C:\code> c:\python26\python
>>> ...ten sam kod...
...
u'Learning Python'
u'Programming Python'
u'Python Pocket Reference'

```

Programy, które muszą zajmować się wynikami analizy składniowej na nietypowe sposoby, będą musiały uwzględnić zmieniony typ obiektu z Pythona 3.x. I znów, ponieważ wszystkie łańcuchy znaków obsługują prawie identyczne interfejsy w wersjach 2.x oraz 3.x, na większość skryptów zmiana ta nie będzie miała żadnego wpływu. Narzędzia dostępne dla obiektów `unicode` z Pythona 2.x są zazwyczaj swobodnie dostępne dla obiektów `str` z wersji 3.x. Najważniejszą kwestią jest stosowanie właściwych nazw standardów kodowania w metodach, które przeanalizowane dane odczytują i zapisują w plikach, przesyłają przez sieć, wyświetlają w interfejsie graficznym lub wykonują inne operacje.

Niestety, dalsze zagłębianie się w szczegółły analizy składniowej XML wykracza poza zakres niniejszej książki. Osoby zainteresowane analizą składniową tekstu oraz kodu XML odsyłam do bardziej szczegółowego omówienia tego zagadnienia w książce poświęconej dziedzinie aplikacji — *Programming Python*. Więcej informacji na temat modułów `re`, `struct`, `pickle` oraz ogólnie narzędzi do obsługi XML można znaleźć w internecie, wspomnianej wyżej książce, innych publikacjach oraz dokumentacji biblioteki standardowej Pythona.

Dlaczego jest to ważne? Badanie plików i nie tylko

Gdy aktualizowałem ten rozdział, natknąłem się na problem z nietypowym działaniem jednego z opisanych narzędzi. Po otwarciu w Notatniku pliku HTML zawierającego wyłącznie znaki ASCII, a następnie zapisaniu go w formacie UTF-8 okazało się, że z powodu uszkodzenia klawiatury pojawił się w nim dziwny znak spoza zakresu ASCII. Z tego powodu nie można było tego pliku przetwarzać za pomocą narzędzi przystosowanych do tekstu w formacie ASCII. Aby znaleźć błędny znak, napisałem w Pythonie prosty kod, który otwierał plik w trybie tekstowym UTF-8, odczytywał plik znak po znaku i zatrzymywał się na pierwszym spoza zestawu ASCII:

```

>>> f = open('py33-windows-launcher.html', encoding='utf8')
>>> t = f.read()
>>> for (i, c) in enumerate(t):
    try:
        x = c.encode(encoding='ascii')
    except:
        print(i, sys.exc_info()[0])

```

```
9886 <class 'UnicodeEncodeError'>
```

Po uzyskaniu indeksu znaku mogłem łatwo wyodrębnić ciąg Unicode, aby uzyskać bardziej szczegółowe informacje:

```
>>> len(t)
```

```
31021
```

```
>>> t[9880:9890]
```

```
'ugh. \u206cThi'
```

```
>>> t[9870:9890]
```

```
'trace through. \u206cThi'
```

Po poprawieniu pliku otworzyłem go w trybie binarnym, aby dokładniej sprawdzić jego oryginalną zawartość:

```
>>> f = open('py33-windows-launcher.html', 'rb')
```

```
>>> b = f.read()
```

```
>>> b[0]
```

```
60
```

```
>>> b[:10]
```

```
b '<HTML>\r\n<T'
```

Nie jest to jakieś zaawansowane podejście i na pewno są lepsze sposoby, ale jak widać, Python oferuje wygodne w użyciu narzędzia na wypadki takie jak powyższy, a obiekty plikowe, użyte czy to w skrypcie, czy w trybie interaktywnym, dają szczegółowy wgląd w zawartość plików.

Osoby chcące zapoznać się z bardziej praktycznymi przykładami przetwarzania danych Unicode zachęcam do lektury innej mojej książki, *Programming Python* (wydanie czwarte lub nowsze). Opisane są w niej przykłady o wiele bardziej rozbudowane niż przedstawione w tym rozdziale, często dotykające kodowania Unicode w kontekście operacji na plikach, przeglądania katalogów, przesyłania danych przez sieć, tworzenia interfejsów graficznych, przetwarzania nagłówków i treści wiadomości e-mail, zawartości stron WWW, korzystania z baz danych i wielu innych zagadnień. Znaczenie kodowania Unicode, które oczywiście jest ważne w świecie globalnego oprogramowania, jest większe, niż się na pozór wydaje. Jest to szczególnie widoczne w Pythonie 3.x, w którym Unicode awansowało do podstawowego kodowania ciągów znaków i plików tekstowych, przez co każdy użytkownik musi je poznać, czy tego chce, czy nie!

Poruszonego tu zagadnienia dotyczy również przedstawiony w rozdziale 9. przykład przetwarzania danych w formacie *JSON*. Jest to niezależny od języka format wymiany danych, którego struktura jest bardzo podobna do słownika lub listy w Pythonie. Ciągi znaków są zawsze kodowane w formacie Unicode, który jest inaczej traktowany w wersjach Pythona 2.x i 3.x, podobnie jak opisany tutaj format XML.

Podsumowanie rozdziału

Niniejszy rozdział omawiał zaawansowane typy łańcuchów znaków dostępne w Pythonie 3.x oraz 2.x i służące do przetwarzania tekstu Unicode oraz danych binarnych. Jak widzieliśmy, wielu programistów wykorzystuje tekst ASCII — dla nich wystarczający będzie podstawowy typ

łańcucha znaków oraz jego wbudowane operacje. W przypadku zastosowań bardziej zaawansowanych model łańcuchów znaków Pythona w pełni obsługuje zarówno tekst Unicode (w wersji 3.x za pomocą normalnego typu łańcucha znaków, w wersji 2.x za pomocą specjalnego typu), jak i dane bajtowe (reprezentowane w wersji 3.x za pomocą typu bytes, a w wersji 2.x — normalnych łańcuchów znaków).

Dodatkowo zobaczyliśmy, w jaki sposób obiekt pliku Pythona został w wersji 3.x przekształcony w taki sposób, by automatycznie kodować i dekodować tekst Unicode i radzić sobie z łańcuchami bajtowymi w plikach w trybie binarnym. Wreszcie krótko omówiliśmy niektóre narzędzia do obsługi danych tekstowych oraz binarnych z biblioteki Pythona, a także wypróbowaliśmy ich działanie w wersjach 3.x i 2.x.

W kolejnym rozdziale skoncentrujemy się na zagadnieniach dotyczących budowy narzędzi, przyglądając się sposobom zarządzania dostępem do atrybutów obiektów za pomocą wstawiania automatycznie wykonywanego kodu. Zanim jednak przejdziemy dalej, poniżej znajduje się zbiór pytań sprawdzających to, czego dowiedzieliśmy się tutaj.

Sprawdź swoją wiedzę — quiz

1. Jakie są nazwy i role typów obiektów łańcuchów znaków z Pythona 3.x?
2. Jakie są nazwy i role typów obiektów łańcuchów znaków z Pythona 2.x?
3. W jaki sposób można na siebie odwzorować typy łańcuchów znaków z Pythona 2.x oraz 3.x?
4. W jaki sposób typy łańcuchów znaków z Pythona 3.x różnią się od siebie w zakresie obsługiwanych operacji?
5. W jaki sposób można zapisać w kodzie znaki Unicode spoza zakresu ASCII w łańcuchu znaków Pythona 3.x?
6. Jaka są główne różnice pomiędzy plikami w trybie tekstowym a binarnym w Pythonie 3.x?
7. W jaki sposób można wczytać plik z tekstem Unicode zawierający tekst z kodowaniem innym od domyślnego dla naszej platformy?
8. W jaki sposób można utworzyć plik tekstowy Unicode z określonym formatem kodowania?
9. Dlaczego tekst ASCII uznawany jest za rodzaj tekstu Unicode?
10. Jak duży wpływ zmiana typów łańcuchów znaków z Pythona 3.x ma na nasz kod?

Sprawdź swoją wiedzę — odpowiedzi

1. Python 3.x zawiera trzy typy łańcuchów znaków: str (przeznaczony dla tekstu Unicode, w tym ASCII), bytes (dla danych binarnych z bezwzględnymi wartościami bajtów) oraz bytearray (zmienną odmianę typu bytes). Typ str zazwyczaj reprezentuje zawartość przechowywaną w plikach tekstowych, natomiast pozostałe dwa typy reprezentują zawartość przechowywaną w plikach binarnych.

2. Python 2.x zawiera dwa podstawowe typy łańcuchów znaków: `str` (przeznaczony dla tekstu 8-bitowego oraz danych binarnych) oraz `unicode` (tekst Unicode). Typ `str` wykorzystywany jest zarówno w przypadku zawartości plików tekstowych, jak i binarnych; typ `unicode` wykorzystywany jest dla zawartości plików, która jest bardziej złożona od 8 bitów. Python 2.x (ale wersje wcześniejsze nie) zawiera również typ `bytearray` z wersji 3.x, jednak został on tam przeniesiony z nowszej wersji i nie przejawia tak ostrego rozróżnienia pomiędzy tekstem a danymi binarnymi, jak ma to miejsce w Pythonie 3.x.
3. Odwzorowanie z typów łańcuchów znaków Pythona 2.x na typy z wersji 3.x nie jest bezpośrednie, ponieważ typ `str` z 2.x równy jest typom `str` i `bytes` z 3.x, a typ `str` z 3.x równy jest typom `str` i `unicode` z 2.x. Zmienność typu `bytearray` z Pythona 3.x jest unikalna. Podsumowując: kodowanie Unicode obsługuje w wersji 3.x typ `str`, a w wersji 2.x typ `unicode`. Natomiast dane binarne i prostsze dane tekstowe obsługuje w wersji 3.x typ `bytes`, a w wersji 2.x typ `str`.
4. Typy łańcuchów znaków Pythona 3.x współdzielą prawie wszystkie operacje — wywołania metod, działania na sekwencjach, a nawet większe narzędzia, takie jak dopasowywanie wzorców, działają w ten sam sposób. Z drugiej strony, jedynie typ `str` obsługuje operacje formatowania łańcuchów znaków, a typ `bytearray` ma zbiór dodatkowych operacji wykonujący modyfikacje w miejscu. Typy `str` oraz `bytes` mają także metody służące, odpowiednio, do kodowania i dekodowania tekstu.
5. Znaki spoza zakresu ASCII można zapisać w kodzie łańcucha znaków za pomocą sekwencji ucieczki, zarówno szesnastkowych (`\xNN`), jak i Unicode (`\uUNNNN`, `\UNNNNNNNN`). Na niektórych klawiaturach pewne znaki spoza zakresu ASCII — na przykład niektóre znaki z kodowania Latin-1 — można także wpisać w sposób bezpośredni. Użyte w skrypcie są domyślnie traktowane jako zakodowane w formacie UTF-8 lub innym, określonym w dyrektywie w komentarzu.
6. W Pythonie 3.x pliki w trybie tekstowym zakładają, że ich zawartość jest tekstem Unicode (nawet jeśli tekst ten jest w kodowaniu ASCII) i automatycznie są one dekodowane przy wczytywaniu oraz kodowane przy zapisywaniu. W przypadku plików w trybie binarnym bajty są przenoszone do pliku i z niego bez zmian. Zawartość plików tekstowych zazwyczaj reprezentowana jest w skrypcie jako obiekty `str`, natomiast zawartość plików binarnych jako obiekty `bytes` (lub `bytearray`). Pliki w trybie tekstowym obsługują również sekwencję znacznika kolejności bajtów BOM dla pewnych typów kodowania i dokonują automatycznie translacji sekwencji końca wiersza z pojedynczego znaku `\n` i na niego na wejściu i wyjściu, o ile opcja ta nie jest w sposób jawnny wyłączona. Pliki w trybie binarnym nie wykonują żadnego z tych kroków. W Pythonie 2.x do otwierania plików Unicode służy metoda `codecs.open`, która w podobny sposób koduje i dekoduje znaki. Natomiast metoda `open` przekształca jedynie końce wierszy w pliku otwartym w trybie tekstowym.
7. W celu wczytania plików zakodowanych w kodowaniu innym od domyślnego dla naszej platformy wystarczy przekazać nazwę kodowania pliku do wbudowanej funkcji `open` Pythona 3.x (w wersji 2.x — `codecs.open()`). Po wczytaniu dane zostaną zdekodowane zgodnie z określonym schematem kodowania. Można również wczytać pliki w trybie binarnym i ręcznie zdekodować bajty na łańcuch znaków, podając nazwę kodowania, jednak wymaga to większego nakładu pracy i jest nieco bardziej podatne na błędy w przypadku znaków wielobajtowych (możemy przypadkowo wczytać część sekwencji znaków).
8. W celu utworzenia pliku tekowego Unicode o określonym formacie kodowania należy w Pythonie 3.x przekazać pożądaną nazwę kodowania do funkcji `open` (w wersji 2.x — do `codecs.open()`). Przy zapisie do pliku łańcuchy znaków zostaną zakodowane zgodnie z podanym schematem. Można również ręcznie zakodować

łańcuch znaków na bajty i zapisać go w trybie binarnym, jednak wymaga to większego nakładu pracy.

9. Tekst ASCII uznawany jest za typ tekstu Unicode, ponieważ jego 7-bitowy przedział wartości jest podzbiorem większości schematów kodowania Unicode. Przykładowo poprawny tekst ASCII jest także poprawnym tekstem Latin-1 (Latin-1 po prostu przypisuje pozostałe możliwe wartości w 8-bitowym bajcie dodatkowym znakom) oraz poprawnym tekstem UTF-8 (UTF-8 definiuje schemat o zmiennej liczbie bajtów reprezentujący więcej znaków, jednak znaki ASCII nadal reprezentowane są przez te same kody, w pojedynczym bajcie). Dzięki temu kodowanie Unicode jest kompatybilne wstecz ze wszechobecnym standardem ASCII (z tego powodu ma jednak pewne mankamenty, na przykład trudno jest automatycznie identyfikować format, dlatego został wprowadzony znacznik BOM).
10. Wpływ zmian modelu łańcuchów znaków z Pythona 3.x uzależniony jest od wykorzystywanych łańcuchów znaków. W przypadku skryptów wykorzystujących prosty tekst ASCII najprawdopodobniej nie ma to żadnego znaczenia — typ łańcucha znaków `str` działa w tym przypadku tak samo w wersjach 2.x oraz 3.x. Co więcej, choć narzędzia biblioteki standardowej wykorzystujące łańcuchy znaków, takie jak moduły `re`, `struct`, `pickle` oraz `xml`, mogą w wersji 3.x korzystać z innych typów niż w Pythonie 2.x, zmiany są dla większości programów bez znaczenia, ponieważ typy `str` i `bytes` z Pythona 3.x oraz typ `str` z 2.x obsługują prawie identyczne interfejsy. Jeśli przetwarzamy dane Unicode, potrzebny nam zbiór narzędzi przeniesiony został z `unicode` i `codecs.open()` Pythona 2.x do `str` i `open` wersji 3.x. Jeśli mamy do czynienia z plikami danych binarnych, musimy obsługiwać ich zawartość w postaci obiektów `bytes`. Ponieważ jednak mają one interfejs podobny do łańcuchów znaków z wersji 2.x, wpływ tych zmian powinien być minimalny. Dlatego w nowym wydaniu książki *Programming Python*, dostosowanym do Pythona 3.x, opisanych jest wiele przykładów uwzględniających zmiany, jakie kodowanie Unicode przyjęte jako obowiązujące w tej wersji standard wprowadziło w interfejsach wielu bibliotek przeznaczonych m.in. do wykonywania operacji sieciowych, tworzenia interfejsów graficznych, korzystania z baz danych i przetwarzania wiadomości e-mail. Zmiany te dotkną prawdopodobnie większość programistów.

Rozdział 38. Zarządzane atrybuty

Niniejszy rozdział rozszerza informacje na temat zaprezentowanych wcześniej technik *przechwytywania atrybutów*, wprowadza kolejne i wykorzystuje je w kilku większych przykładach. Tak jak wszystkie pozostałe rozdziały z tej części książki, został on zaklasyfikowany do tematów bardziej zaawansowanych i opcjonalnych, ponieważ większość programistów aplikacji nie musi się przejmować omawianymi tutaj zagadnieniami — mogą oni pobierać i ustawiać atrybuty obiektów bez martwienia się o samą implementację atrybutów.

Zarządzanie dostępem do atrybutów może jednak być szczególnie istotne dla twórców narzędzi — jako część elastycznego API. Co więcej, znajomość opisanego tutaj modelu deskryptorów pozwoli lepiej zrozumieć inne narzędzia, takie jak sloty i właściwości, a nawet może okazać się niezbędna, jeżeli narzędzia te zostaną użyte w wykorzystywanym kodzie.

Po co zarządza się atrybutami

Atrybuty obiektów są kluczowym elementem większości programów napisanych w Pythonie — to w nich często przechowujemy informacje o jednostkach przetwarzanych przez nasz skrypt. Normalnie atrybuty są po prostu zmiennymi obiektów — atrybut `name` osoby może na przykład być prostym łańcuchem znaków, pobieranym i ustawianym za pomocą podstawowej składni atrybutów:

```
person.name          # Pobranie wartości atrybutu  
person.name = wartość # Modyfikacja wartości atrybutu
```

W większości przypadków atrybut znajduje się w samym obiekcie lub jest dziedziczony po klasie, z której obiekt ten pochodzi. Ten podstawowy model będzie wystarczający na potrzeby większości programów pisanych w trakcie naszej kariery programisty Pythona.

Czasami jednak wymagana jest większa elastyczność. Przypuśćmy, że napisaliśmy program korzystający z atrybutu `name` w sposób bezpośredni, jednak później wymagania się zmieniają — na przykład decydujemy o tym, że dane te powinny być w jakiś sposób sprawdzane bądź modyfikowane przy pobraniu. Napisanie kodu metod zarządzającego dostępem do wartości atrybutów jest stosunkowo proste (`valid` i `transform` są tutaj wartościami abstrakcyjnymi).

```
class Person:  
    def getName(self):  
        if not valid():  
            raise TypeError('nie można pobrać danych')  
        else:  
            return self.name.transform()  
    def setName(self, value):
```

```

if not valid(value):
    raise TypeError('nie można zmienić danych')
else:
    self.name = transform(value)
person = Person()
person.getName()
person.setName('value')

```

Taka modyfikacja wymaga jednak wprowadzenia zmian w całym programie — we wszystkich miejscach, w których wykorzystywane są dane osobowe ze zmiennej `name`, co może nie być trywialnym zadaniem. Co więcej, takie rozwiązanie wymaga, by program był świadom sposobu eksportowania wartości — w postaci prostych zmiennych lub wywoływanych metod. Jeśli zaczniemy od interfejsu danych opartego na metodach, klient jest odporny na zmiany. Jeśli tak jednak nie będzie, może się to stać problematyczne.

Takie problemy pojawiają się częściej, niż można by tego oczekiwać. Wartość komórki w programie przypominającym arkusz kalkulacyjny może na przykład rozpocząć swój żywot jako prosta, niezależna wartość, ale później może ona przemienić się w dowolne obliczenia. Ponieważ interfejs obiektu powinien być na tyle elastyczny, by obsługiwać tego typu przyszłe zmiany bez zakłócania działania istniejącego kodu, późniejsze przełączenie się na metody jest dalekie od ideału.

Wstawianie kodu wykonywanego w momencie dostępu do atrybutów

Lepszym rozwiązaniem byłoby pozwolenie na automatyczne wykonywanie kodu w momencie dostępu do atrybutu, jeśli jest to potrzebne. Jest to jedna z głównych cech zarządzanego atrybutu — umożliwia on definiowanie kodu, który jest wykonywany po uzyskaniu dostępu. Uogólniając, taki atrybut działa w trybie daleko wykraczającym poza proste przechowywanie danych.

W różnych miejscach książki spotkaliśmy narzędzia Pythona pozwalające skryptom automatycznie obliczać wartości atrybutów przy ich pobieraniu i sprawdzające poprawność lub modyfikujące wartości atrybutów przy przechowywaniu. W niniejszym rozdziale rozszerzymy informacje na temat wprowadzonych już narzędzi, poznamy inne dostępne, a także przyjrzymy się większym przykładom przypadków użycia z tej dziedziny. W szczególności rozdział ten prezentuje:

- Metody `__getattribute__` oraz `__setattr__` służące do przekierowywania niezdefiniowanych pobrań atrybutów, a także wszystkich przypisań atrybutów do ogólnych metod programów obsługi.
- Metodę `__getattribute__` służącą do przekierowywania wszystkich pobrań atrybutów do ogólnej metody programu obsługi.
- Wbudowaną funkcję `property` służącą do przekierowywania dostępu do uszczegółowionych atrybutów do funkcji programów obsługi pobierania i ustawiania.
- *Protokół deskryptora* służący do przekierowywania dostępu do uszczegółowionych atrybutów do instancji klas z własnymi metodami programów obsługi pobierania i ustawiania, stanowiącymi bazę dla innych narzędzi, takich jak właściwości i sloty.

Narzędzia wymienione w pierwszym punkcie są dostępne we wszystkich wersjach Pythona, natomiast te z pozostałych trzech są dostępne w wersji 3.x oraz nowych klasach w wersji 2.x. Klasyczne te najpierw pojawiły się w wersji 2.2 wraz innymi zaawansowanymi narzędziami, np.

slotami i funkcją `super`, opisanymi w rozdziale 32. Narzędzia wymienione w punktach pierwszym i trzecim zostały ogólnie opisane odpowiednio w rozdziałach 30. i 32., natomiast zagadnienia z punktów drugiego i czwartego to nowości, szczegółowo opisane w tym rozdziale.

Jak zobaczymy, wszystkie cztery techniki do pewnego stopnia mają wspólne cele i zazwyczaj można rozwiązać określony problem w kodzie za pomocą dowolnej z nich. Istnieją jednak między nimi pewne istotne różnice. Przykładowo dwie ostatnie z wymienionych technik mają zastosowanie do *uszczegółowionych* atrybutów, natomiast dwie pierwsze są na tyle ogólne, by móc je wykorzystywać w klasach opartych na delegacji, które muszą przekierowywać *dowolne* atrybuty do opakowanych obiektów. Jak będziemy mogli się przekonać, wszystkie cztery rozwiązania różnią się także stopniem skomplikowania oraz estetyką, co trzeba zobaczyć w praktyce, by móc dokonać ich samodzielnej oceny.

Poza omówieniem szczegółów leżących u podstaw czterech wymienionych wyżej technik przechwytywania atrybutów w niniejszym rozdziale będziemy mieli okazję zapoznać się z programami większymi od widzianych wcześniej w książce. Studium przypadku `CardHolder` z końca rozdziału powinno służyć jako przykład działania większej klasy do samodzielnego studiowania. Części technik wykorzystanych tutaj użyjemy również w kolejnym rozdziale, w kodzie dekoratorów, dlatego przed przejściem dalej należy upewnić się, że omawiane tutaj zagadnienia zrozumiałe się przynajmniej w ogólnym zarysie.

Właściwości

Protokół właściwości pozwala przekierowywać operacje pobierania i ustawiania określonego atrybutu do udostępnianych przez nas funkcji lub metod, pozwalając nam wstawać kod, który zostanie wykonany automatycznie w momencie dostępu do atrybutów, a także przechwytywać usuwanie atrybutów i udostępniać ich dokumentację, jeśli jest to pożądane.

Właściwości tworzone są za pomocą funkcji wbudowanej `property` i przypisywane są do atrybutów klas, podobnie jak funkcje metod. Tym samym są także dziedziczone przez klasy podzielne oraz instancje, podobnie jak wszystkie inne atrybuty klas. Do ich przechwytyujących dostęp funkcji przekazywany jest argument instancji `self`, który umożliwia dostęp do informacji o stanie i do atrybutów klas dostępnych w podmiotowej instancji.

Właściwość zarządza pojedynczym, określonym atrybutem. Choć nie jest w stanie przechwytywać wszystkich dostępów do atrybutów w sposób ogólny, pozwala na kontrolowanie dostępu związanego z pobieraniem i przypisaniem, a także umożliwia zmianę atrybutu z prostych danych na swobodnie obliczany bez zakłócania działania istniejącego kodu. Jak zobaczymy, właściwości są mocno powiązane z deskryptorami — są właściwie ich ograniczoną postacią.

Podstawy

Właściwość tworzona jest za pomocą przypisania wyniku wbudowanej funkcji do atrybutu klasy:

```
attribute = property(fget, fset, fdel, doc)
```

Żaden z argumentów funkcji wbudowanej nie jest wymagany i w razie ich nieprzekazania wszystkie otrzymują wartość domyślną `None`. Takie operacje nie będą wtedy obsługiwane, a próba ich wykonania spowoduje zwrocenie wyjątku `AttributeError`.

Przy użyciu do `fget` przekazujemy funkcję służącą do przechwytywania pobrania atrybutów, do `fset` — funkcję do przypisania, do `fdel` — funkcję do usuwania atrybutów. Z technicznego punktu widzenia we wszystkich trzech argumentach można umieszczać dowolne funkcje i metody, przy czym pierwszym argumentem takiej funkcji musi być referencja do instancji klasy

posiadającej dany atrybut. Funkcja umieszczona w argumencie fget musi zwracać wartość atrybutu, natomiast funkcje umieszczone w argumentach fset i fdel nie mogą zwracać żadnego wyniku (albo zwracać wartość None). Wszystkie trzy rodzaje funkcji mogą zgłaszać wyjątki oznaczające odmowę dostępu do atrybutu.

Argument doc otrzymuje łańcuch znaków dokumentacji dla atrybutu, jeśli jest to pożądane (w przeciwnym razie właściwość kopiuje łańcuch znaków dokumentacji fget, o ile jest on podany, który ma wartość domyślną None).

Wbudowana funkcja property zwraca obiekt właściwości przypisywany do nazwy atrybutu, którym chcemy zarządzać w zakresie klasy, gdzie zostanie on odziedziczony przez wszystkie instancje.

Pierwszy przykład

W celu zademonstrowania, jak przekłada się to na działający kod, poniższa klasa wykorzystuje właściwość do śledzenia dostępu do atrybutu o nazwie name. Same przechowywane dane noszą nazwę named_name w celu uniknięcia konfliktu nazw z właściwością. (Informacja dla wykonujących opisane w tej książce przykłady: pod listingami znajdują się polecenia z nazwami plików, uruchamiające opisywane kody).

```
class Person:                                     # W wersji 2.x należy użyć (object)
    def __init__(self, name):
        self._name = name
    def getName(self):
        print('pobieranie...')
        return self._name
    def setName(self, value):
        print('modyfikacja...')
        self._name = value
    def delName(self):
        print('usunięcie...')
        del self._name
    name = property(getName, setName, delName, "Dokumentacja właściwości
name")
bob = Person('Robert Zielony')                  # bob ma zarządzany atrybut
print(bob.name)                                 # Wykonuje getName
bob.name = 'Robert A. Zielony'                 # Wykonuje setName
print(bob.name)
del bob.name                                    # Wykonuje delName
print('*'*20)
anna = Person('Anna Czerwona')                 # anna także dziedziczy właściwość
print(anna.name)
```

```
print(Person.name.__doc__) # Lub help(Person.name)
```

Właściwości dostępne są w Pythonie 2.x oraz 3.x, jednak w wersji 2.x wymagają pochodzenia obiektów w nowym stylu, by przypisania działały poprawnie. By wykonać powyższy kod w Pythonie 2.x, należy dodać `object` jako klasę nadziedzianą (w wersji 3.x również można dodać klasę nadziedzianą, jednak jest to domyślne i nie jest wymagane).

Ta akurat właściwość niewiele robi — po prostu przechwytuje i śledzi atrybut — jednak służy do zademonstrowania protokołu. Kiedy powyższy kod zostanie wykonany, dwie instancje dziedziczą właściwość — w ten sam sposób jak odziedziczyłyby dowolny inny atrybut dołączony do swojej klasy. Przechwytywane są jednak operacje dostępu do atrybutów:

```
c:\code> py -3 prop-person.py
```

```
pobieranie...
```

```
Robert Zielony
```

```
modyfikacja...
```

```
pobieranie...
```

```
Robert Zielony
```

```
usunięcie...
```

```
-----
```

```
pobieranie...
```

```
Anna Czerwona
```

```
Dokumentacja właściwości name
```

Tak jak wszystkie atrybuty klas, właściwości są *dziedziczony* zarówno przez instancje, jak i klasy podzielone znajdujące się niżej w hierarchii. Jeśli zmodyfikujemy nasz przykład w następujący sposób:

```
class Super:  
    ...kod oryginalnej klasy Person...  
  
    name = property(getName, setName, delName, 'Dokumentacja właściwości  
name')  
  
class Person(Super):  
    pass # Właściwości są dziedziczone  
  
bob = Person('Robert Zielony')  
...reszta bez zmian...
```

Wynik będzie taki sam — klasa podzielona `Person` dziedziczy właściwość `name` po klasie `Super`, natomiast instancja `bob` otrzymuje ją po `Person`. Jeśli chodzi o dziedziczenie, właściwości działają tak samo jak normalne metody; ponieważ mają dostęp do argumentów instancji `self`, mogą uzyskiwać dostęp do informacji o stanie w instancji, podobnie jak metody, co zobaczymy niżej.

Obliczanie atrybutów

Przykład powyżej śledzi po prostu dostęp do atrybutów. Zazwyczaj jednak właściwości robią o wiele więcej — na przykład obliczają wartość atrybutu w sposób dynamiczny w momencie

pobierania. Poniższy przykład ilustruje takie zastosowanie:

```
class PropSquare:  
    def __init__(self, start):  
        self.value = start  
    def getX(self):                      # Przy pobraniu atrybutów  
        return self.value ** 2  
    def setX(self, value):                # Przy przypisaniu atrybutów  
        self.value = value  
    X = property(getX, setX)             # Brak usuwania i dokumentacji  
P = PropSquare(3)                     # 2 instancje klasy z właściwością  
Q = PropSquare(32)                    # Każda ma inne informacje o stanie  
print(P.x)                           # 3 ** 2  
P.x = 4  
print(P.x)                           # 4 ** 2  
print(Q.x)                           # 32 ** 2
```

Powyższa klasa definiuje atrybut X, do którego dostęp odbywa się tak, jakby był on danymi statycznymi, jednak tak naprawdę wykonuje kod obliczający jego wartość przy pobraniu. Rezultat przypomina niejawne wywołanie metody. Kiedy kod jest wykonywany, wartość przechowywana jest w instancji w postaci informacji o stanie, jednak za każdym razem, gdy pobieramy ją za pomocą zarządzanego atrybutu, wartość ta jest automatycznie podnoszona do kwadratu.

```
c:\code> py -3 prop-computed.py  
9  
16  
1024
```

Warto zwrócić uwagę na to, że utworzyliśmy dwie różne instancje. Ponieważ metody właściwości automatycznie otrzymują argument self, mają dostęp do informacji o stanie przechowywanych w instancjach. W naszym przypadku oznacza to, że operacja pobrania oblicza kwadrat danych podmiotowej instancji.

Zapisywanie właściwości w kodzie za pomocą dekoratorów

Choć dodatkowe szczegóły zachowamy na kolejny rozdział, podstawy dekoratorów funkcji wprowadziliśmy wcześniej, w rozdziale 32. Przypomnijmy, że składnia dekoratorów:

```
@dekorator  
def funkcja(argumenty): ...
```

automatycznie przekładana jest przez Pythona na poniższy odpowiednik w celu ponownego dowiezania nazwy funkcji do wyniku obiektu wywoływalnego *dekorator*:

```
def funkcja(argumenty): ...
    funkcja = dekorator(funkcja)
```

Dzięki temu odwzorowaniu okazuje się, że wbudowana funkcja `property` może służyć jako dekorator, definiujący funkcję, która zostanie wykonana automatycznie, kiedy atrybut zostaje pobrany:

```
class Person:
    @property
    def name(self): ... # Ponownie dowiązuje name = property(name)
```

Po wykonaniu do udekorowanej metody automatycznie przekazywany jest pierwszy argument funkcji wbudowanej `property`. Jest to tak naprawdę składnia alternatywna dla tworzenia właściwości i ręcznego, ponownego dowiązania nazwy atrybutu.

```
class Person:
    def name(self): ...
    name = property(name)
```

Dekoratory setter i deleter

Od Pythona 2.6 i 3.0 obiekty właściwości mogą zawierać metody `getter`, `setter` i `deleter`, które przypisują odpowiednio metody akcesora właściwości i zwracają kopię samej właściwości. Możemy je wykorzystać do określenia komponentów właściwości, dekorując także normalne metody, choć komponent `getter` zazwyczaj wypełniany jest automatycznie przez sam fakt tworzenia właściwości.

```
class Person:
    def __init__(self, name):
        self._name = name
    @property
    def name(self): # name = property(name)
        "Dokumentacja właściwości name"
        print('pobieranie...')
        return self._name
    @name.setter
    def name(self, value): # name = name.setter(name)
        print('modyfikacja...')
        self._name = value
    @name.deleter
    def name(self): # name = name.deleter(name)
        print('usunięcie...')
        del self._name
bob = Person('Robert Zielony') # bob ma zarządzany atrybut
```

```

print(bob.name)                                # Wykonuje komponent getter dla name
(pierwszy dostęp do name)

bob.name = 'Robert A. Zielony'                 # Wykonuje komponent setter dla name
(drugi dostęp do name)

print(bob.name)

del bob.name                                    # Wykonuje komponent deleter dla name
(trzeci dostęp do name)

print('*'*20)

anna = Person('Anna Czerwona')                 # anna także dziedziczy właściwość
print(anna.name)

print(Person.name.__doc__)                      # Lub help(Person.name)

```

Tak naprawdę powyższy kod jest odpowiednikiem pierwszego przykładu z tego podrozdziału — dekoracja to w tym przypadku po prostu alternatywny sposób zapisania w kodzie właściwości. Po wykonaniu wyniki są takie same.

```

c:\code> py -3 prop-person-deco.py
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja właściwości name

```

W porównaniu z ręcznym przypisywaniem wyników za pomocą `property` w tym przypadku użycie dekoratorów do tworzenia właściwości wymaga jedynie trzech dodatkowych wierszy kodu (różnica jest do pominięcia). Jak to często bywa w przypadku narzędzi alternatywnych, wybór pomiędzy dwoma technikami jest w dużej mierze kwestią subiektywną.

Deskryptory

Deskryptory stanowią alternatywny sposób przechwytywania dostępu do atrybutów i są mocno powiązane z właściwościami omówionymi w poprzednim podrozdziale. Tak naprawdę właściwość jest rodzajem deskryptora — funkcja wbudowana `property` jest uproszczonym sposobem tworzenia określonego typu deskryptora, który wykonuje funkcje metod w momencie dostępu do atrybutów. Deskryptory implementują mechanizm wykorzystywany przez różne narzędzia w klasie, w tym właściwości i sloty.

Z punktu widzenia funkcjonalności protokół deskryptora pozwala nam przekierować operacje pobierania i ustawiania określonego atrybutu do metod osobnego obiektu klasy, który

udostępnimy. Umożliwiają one wstawienie kodu wykonywanego automatycznie w momencie dostępu do atrybutu i pozwalają przechwytywać operacje usuwania atrybutów, a także udostępniać dokumentację, jeśli jest to pożądane.

Deskryptory tworzone są jako niezależne *klasy* i są przypisywane do atrybutów klas tak samo jak funkcje metod. Tak jak wszystkie inne atrybuty klas, są one dziedziczone przez klasy podzielone oraz instancje. Do ich metod przechwytyujących operacje dostępu przekazywane są zarówno sam deskryptator (w postaci argumentu `self`), jak i instancje klasy klienta. Z tego powodu zachowują i wykorzystują własne informacje o stanie, a także informacje o stanie podmiotowej instancji. Przykładowo deskryptator może wywoływać metody dostępne w klasie klienta, a także definiowane przez niego metody specyficzne dla deskryptora.

Podobnie jak właściwość, deskryptor zarządza pojedynczym, określonym atrybutem. Choć nie jest w stanie przechwytywać wszystkich dostępów do atrybutów w sposób uniwersalny, udostępnia kontrolę nad dostępem związanym zarówno z pobieraniem, jak i z przypisywaniem, a także pozwala dowolnie modyfikować atrybut z prostych danych na obliczenia bez zakłócania działania istniejącego kodu. Właściwości tak naprawdę są po prostu wygodnym sposobem tworzenia określonego typu deskryptora i, jak zobaczymy, można je tworzyć w kodzie bezpośrednio w postaci deskryptorów.

Podczas gdy właściwości mają stosunkowo wąski zakres, deskryptory są rozwiązaniem bardziej uniwersalnym. Przykładowo, ponieważ tworzone są w postaci normalnych klas, deskryptory mają swój własny stan, mogą brać udział w hierarchiach dziedziczenia deskryptorów, wykorzystują kompozycję do agregacji obiektów i stanowią naturalną strukturę dla tworzenia w kodzie metod wewnętrznych oraz łańcuchów znaków dokumentacji atrybutów.

Podstawy

Jak wspomniano wcześniej, deskryptory zapisywane są w kodzie jako odrębne klasy i udostępniają metody akcesorów o specjalnych nazwach, służące do operacji dostępu do atrybutów, które mają przechwytywać. Metody pobierania (`__get__`), ustawiania (`__set__`) oraz usuwania (`__delete__`) w klasie deskryptora są wykonywane automatycznie w momencie wystąpienia odpowiedniego typu dostępu do atrybutu przypisanego do instancji klasy deskryptora.

```
class Descriptor:
    "miejsce na łańcuch znaków dokumentacji"
    def __get__(self, instancja, właściwie):
        # Zwraca wartość atrybutu
    def __set__(self, instancja, właściwie):
        # Nic nie zwraca (None)
    def __delete__(self, instancja):
        # Nic nie zwraca (None)
```

Klasy zawierające dowolną z powyższych metod uznawane są za deskryptory, a ich metody są specjalne, jeśli jedna z ich instancji zostanie przypisana do atrybutu innej klasy — przy dostępie do atrybutu są one wywoływane automatycznie. Jeśli któraś z metod jest nieobecna, oznacza to zazwyczaj, że odpowiadający jej typ dostępu nie jest obsługiwany. W przeciwnieństwie do właściwości pominięcie metody `__set__` pozwala na ponowne zdefiniowanie nazwy w instancji, tym samym ukrywając deskryptor. By atrybut był *tylko do odczytu*, należy zdefiniować metodę `__set__` w taki sposób, aby przechwytywała przypisania i zgłaszała wyjątek.

Deskryptor zawierający metodę `__set__` wywołuje pewne specyficzne implikacje podczas dziedziczenia klas. Ten temat został odłożony do rozdziału 40., w którym szczegółowo zostały opisane metaklasy i proces dziedziczenia. W skrócie: deskryptor zawierający metodę `__set__`

jest nazywany *deskryptorem danych* i w regułach zwykłego dziedziczenia opatrzona nim nazwa jest traktowana priorytetowo. Na przykład odziedziczony deskryptor nazwy `__class__` zastępuje nazwę instancji. Ponadto deskryptory kodowane we własnych klasach mają pierwszeństwo przed innymi deskryptorami.

Argumenty metod deskryptorów

Zanim zajmiemy się utworzeniem jakiegoś realistycznego przykładu, przyjrzyjmy się podstawom. Do wszystkich trzech opisanych powyżej metod deskryptorów przekazywana jest zarówno instancja klasy deskryptora (`self`), jak i instancja klasy klienta (`instancja`), do której dołączana jest instancja deskryptora.

Metoda dostępu `__get__` dodatkowo otrzymuje argument *właściciel* określający klasę, do której dołączana jest instancja deskryptora. Jej argument `instancja` jest albo instancją, przez którą odbył się dostęp do atrybutu (dla `instancja.atrybut`), lub `None`, jeśli dostęp do atrybutu odbywa się bezpośrednio za pomocą klasy właściciela (dla `klasa.atrybut`). Pierwsza forma zazwyczaj oblicza wartość dla dostępu do instancji, natomiast druga najczęściej zwraca `self`, jeśli obsługiwany jest dostęp do obiektu deskryptora.

Przykładowo w poniższym kodzie uruchomionym w wersji Pythona 3.x po pobraniu `X.attr` automatycznie jest wykonywana metoda `__get__` klasy `Descriptor`, do której przypisany jest atrybut klasy `Subject.attr`. W wersji 2.x należy użyć odpowiednika instrukcji `print` i utworzyć dwie klasy pochodne od `object`, ponieważ deskryptor jest narzędziem dostępnym w nowego typu klasach. W wersji 3.x dziedziczenie jest wykonywane niejawnie i można je pominąć, choć stosowanie go nie jest błędem:

```
>>> class Descriptor:                                     # W wersji 2.x należy dodać "
   (object)"

...     def __get__(self, instance, owner):
...         print(self, instance, owner, sep='\n')
...
...
>>> class Subject:                                     # W wersji 2.x należy dodać "
   (object)"

...     attr = Descriptor()                           # Instancja klasy Descriptor
jest atrybutem klasy
...
...
>>> X = Subject()
>>> X.attr
<__main__.Descriptor object at 0x0281E690>
<__main__.Subject object at 0x028289B0>
<class '__main__.Subject'>
>>> Subject.attr
<__main__.Descriptor object at 0x0281E690>
None
<class '__main__.Subject'>
```

Warto zwrócić uwagę na argumenty przekazane automatycznie do metody `__get__` przy pierwszym pobraniu atrybutu. Gdy pobierane jest `X.attr`, działa to tak, jakby nastąpił poniższy przekład (choć `Subject.attr` nie wywołuje tutaj ponownie metody `__get__`):

```
X.attr -> Descriptor.__get__(Subject.attr, X, Subject)
```

Deskryptor wie o tym, że odbywa się do niego dostęp bezpośredni, jeśli argument instancji równy jest None.

Deskryptory tylko do odczytu

Jak wspomniano wcześniej, w przeciwieństwie do właściwości, w przypadku deskryptorów pominięcie metody `__set__` nie wystarczy, by atrybut stał się tylko do odczytu, ponieważ zmienną deskryptora można przypisać do instancji. W poniższym kodzie przypisanie atrybutu do `X.a` powoduje przechowanie a w instancji obiektu `X`, tym samym ukrywając deskryptor przechowany w klasie `C`.

```
>>> class D:  
...     def __get__(*args): print('pobranie')  
...  
>>> class C:  
...     a = D()                      # Atrybut a jest instancją deskryptora  
...  
>>> X = C()  
>>> X.a                         # Wykonuje metodę __get__  
odziedziczonego deskryptora  
pobranie  
>>> C.a  
pobranie  
>>> X.a = 99                      # Przechowane w X, ukrywa C.a  
>>> X.a  
99  
>>> list(X.__dict__.keys())  
['a']  
>>> Y = C()  
>>> Y.a                         # Y nadal dziedziczy deskryptor  
pobranie  
>>> C.a  
pobranie
```

W ten sposób działa przypisywanie atrybutów instancji w Pythonie, co pozwala klasom na selektywne nadpisywanie wartości domyślnych z poziomu klasy w ich instancjach. By uczynić atrybut oparty na deskryptorze atrybutem tylko do odczytu, należy przehwndyci przypisanie w klasie deskryptora i zgłosić wyjątek, tak by zapobiec przypisaniu atrybutu. Przy przypisywaniu atrybutu będącego deskryptorem Python obchodzi normalne zachowanie na poziomie instancji i przekierowuje operację do obiektu deskryptora.

```
>>> class D:  
...     def __get__(*args): print('pobranie')
```

```

...     def __set__(*args): raise AttributeError('nie można ustawić')
...
>>> class C:
...     a = D()
...
>>> X = C()
>>> X.a                         # Przekierowane do C.a.__get__
pobranie
>>> X.a = 99                     # Przekierowane do C.a.__set__
AttributeError: nie można ustawić

```



Należy także uważać, by nie pomylić metody deskryptora `__delete__` z ogólną metodą `__del__`. Ta pierwsza wywoływana jest przy próbach usunięcia nazwy zarządzanego atrybutu w instancji klasy właściciela. Ta druga jest ogólną metodą destruktora instancji, wykonywaną gdy instancja klasy dowolnego typu ma być wyczyszczona z pamięci. Metoda `__delete__` jest bliżej związana z ogólną metodą usuwania atrybutu `__delattr__`, z którą spotkamy się w dalszej części rozdziału.Więcej informacji na temat metod przeciążania operatorów można znaleźć w rozdziale 30.

Pierwszy przykład

By zobaczyć, jak to wszystko łączy się ze sobą w bardziej realistycznym kodzie, zacznijmy od tego samego pierwszego przykładu, jaki napisaliśmy dla właściwości. Poniższy kod definiuje deskryptor przechwytyjący dostęp do atrybutu o nazwie `name` w swoich klientach. Jego metody wykorzystują argument instancji w celu uzyskania dostępu do informacji o stanie z podmiotowej instancji, w której przechowywany jest łańcuch znaków imienia i nazwiska. Tak jak właściwości, deskryptory działają poprawnie jedynie dla klas w nowym stylu, dlatego w Pythonie 2.6 trzeba dopilnować pochodzenia *obu* klas od `object`. Dziedziczenie samego deskryptora lub klientów nie wystarczy.

```

class Name:                               # W Pythonie 2.x należy użyć (object)
    "Dokumentacja deskryptora name"
    def __get__(self, instance, owner):
        print('pobieranie...')
        return instance._name
    def __set__(self, instance, value):
        print('modyfikacja...')
        instance._name = value
    def __delete__(self, instance):
        print('usunięcie...')
        del instance._name

```

```

class Person:                                # W Pythonie 2.x należy użyć (object)
    def __init__(self, name):
        self._name = name
        name = Name()                         # Przypisanie deskryptora do atrybutu
    bob = Person('Robert Zielony')           # bob ma zarządzany atrybut
    print(bob.name)                          # Wykonuje Name.__get__
    bob.name = 'Robert A. Zielony'          # Wykonuje Name.__set__
    print(bob.name)
    del bob.name                            # Wykonuje Name.__delete__
    print('*'*20)
    anna = Person('Anna Czerwona')          # anna także dziedziczy deskryptor
    print(anna.name)
    print(Name.__doc__)                    # Lub help(Name)

```

Warto zwrócić uwagę na to, jak w powyższym kodzie przypisujemy instancję klasy deskryptora do *atrybutu klasy* w klasie klienta. Z tego powodu jest on dziedziczony przez wszystkie instancje klasy, tak samo jak jej metody. Tak naprawdę *musimy* przypisać deskryptor do atrybutu klasy w taki właśnie sposób — nie będzie on działał, jeśli zamiast tego przypiszemy go do atrybutu instancji *self*. Po wykonaniu metody *__get__* deskryptora przekazywane są do niej trzy obiekty w celu zdefiniowania kontekstu:

- *self* jest instancją klasy *Name*,
- *instance* jest instancją klasy *Person*,
- *owner* to klasa *Person*.

Po wykonaniu powyższego kodu metody deskryptora przechwytyują próby uzyskania dostępu do atrybutów, podobnie jak wersja kodu z właściwościami. Tak naprawdę wynik będzie znowu taki sam:

```

c:\code> py -3 desc-person.py
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja deskryptora name

```

Podobnie jak w przykładzie z właściwościami, instancja klasy deskryptora jest atrybutem klasy i tym samym *dziedziczona* jest przez wszystkie instancje klasy klienta oraz klasy podzielne. Jeśli

w naszym przykładzie zmienimy klasę Person w następujący sposób, wynik skryptu będzie taki sam:

```
...
class Super:
    def __init__(self, name):
        self._name = name
    name = Name()
class Person(Super):          # Deskryptory są dziedziczone
    pass
...
```

Warto również zauważyć, że kiedy klasa deskryptora nie jest przydatna poza klasą klienta, zupełnie rozsądne jest składniowe osadzenie definicji deskryptora wewnątrz klienta. Oto jak wyglądać będzie nasz przykład w przypadku zastosowania *klasy osadzonej*:

```
class Person:
    def __init__(self, name):
        self._name = name
class Name:                  # Zastosowanie klasy osadzonej
    "Dokumentacja deskryptora name"
    def __get__(self, instance, owner):
        print('pobieranie...')
        return instance._name
    def __set__(self, instance, value):
        print('modyfikacja...')
        instance._name = value
    def __delete__(self, instance):
        print('usunięcie...')
        del instance._name
name = Name()
```

W takim kodzie Name staje się zmienną lokalną w zakresie instrukcji klasy Person i tym samym nie będzie wchodziła w konflikt z żadnymi nazwami spoza klasy. Powyzsza wersja (zawarta w pliku *desc-person-nested.py*) działa tak samo jak oryginalna — przeniesliśmy po prostu definicję klasy deskryptora do zakresu klasy klienta — jednak ostatni wiersz kodu sprawdzającego musi się zmienić, by pobierać łańcuch znaków dokumentacji z nowej lokalizacji.

```
...
print(Person.Name.__doc__)          # Różnica: już nie Name.__doc__
poza klasą
```

Obliczone atrybuty

Tak jak było w przypadku zastosowania właściwości, nasz pierwszy przykład deskryptora z powyższego podrozdziału nie robił nic specjalnego — wyświetlał po prostu komunikaty śledzenia dla dostępu do atrybutów. W praktyce deskryptory można także wykorzystać do obliczania wartości atrybutów za każdym razem, gdy są one pobierane. Ilustruje to poniższy przykład — jest on inną wersją tego samego przykładu, jaki utworzyliśmy dla właściwości. Wykorzystuje on deskryptor do automatycznego obliczenia kwadratu wartości atrybutu z każdym pobraniem.

```
class DescSquare:

    def __init__(self, start):                      # Każdy deskryptor ma własny stan
        self.value = start

    def __get__(self, instance, owner):             # Przy pobieraniu atrybutów
        return self.value ** 2

    def __set__(self, instance, value):              # Przy przypisywaniu atrybutów
        self.value = value                           # Brak usuwania i dokumentacji

class Client1:
    X = DescSquare(3)                            # Przypisanie instancji
    deskryptora do atrybutu klasy

class Client2:
    X = DescSquare(32)                          # Inna instancja w innej klasie
    klienta

    # Można także utworzyć kod dwóch
    # instancji tej samej klasy

c1 = Client1()
c2 = Client2()

print(c1.X)                                    # 3 ** 2
c1.X = 4
print(c1.X)                                    # 4 ** 2
print(c2.X)                                    # 32 ** 2
```

Po wykonaniu wynik powyższego przykładu będzie taki sam jak oryginalnej wersji opartej na właściwościach, jednak tym razem próby dostępu do atrybutów przechwytywane są przez obiekt deskryptora klas.

```
c:\code> py -3 desc-computed.py
9
16
1024
```

Wykorzystywanie informacji o stanie w deskryptorach

Jeśli zastanowimy się chwilę nad dwoma utworzonymi dotychczas przykładami deskryptorów, możemy zauważyć, że swoje informacje pobierają one z różnych miejsc. Pierwszy (przykład z atrybutem name) wykorzystuje dane przechowywane w *instancji* klienta, natomiast drugi

(przykład z kwadratem atrybutu) wykorzystuje dane dołączone do samego obiektu *deskryptora*. Tak naprawdę deskryptory mogą wykorzystywać stan *zarówno* instancji, jak i deskryptora lub też dowolną ich kombinację:

- *Stan deskryptora* wykorzystywany jest do zarządzania danymi wewnętrznymi z punktu widzenia działania deskryptora lub danymi ze wszystkich instancji. Może być inny w każdej klasie klienckiej.
- *Stan instancji* zapisuje informacje powiązane z klasą klienta i prawdopodobnie przez nią utworzone. Może być inny w każdej klasie klienckiej (tj. w każdym obiekcie aplikacyjnym).

Innymi słowy, stan deskryptora to dane charakterystyczne dla deskryptora, a stan instancji to dane charakterystyczne dla instancji klienckiej. Jak zawsze w programowaniu obiektowym należy starannie wybierać stan. Na przykład nie można wykorzystywać stanu *deskryptora* do przechowywania nazwiska pracownika, ponieważ w każdej instancji klienckiej wymagana jest inna wartość. Jeżeli nazwisko zostanie zapisane w stanie deskryptora, wtedy będzie współdzielone przez wszystkie instancje klasy klienckiej. Analogicznie: stanu instancji nie można wykorzystywać do przechowywania wewnętrznych danych deskryptora, ponieważ zostaną utworzone ich różne kopie.

Metody deskryptorów mogą wykorzystywać oba rozwiązania, jednak stan deskryptora często sprawia, że używanie specjalnych konwencji nazewnictwa w celu uniknięcia konfliktów między nazwami danych deskryptora przechowywanych w instancji nie jest konieczne. Poniższy deskryptor dodaje na przykład informacje do własnej instancji, dzięki czemu nie wchodzą one w konflikt z informacjami instancji klasy klienta.

```
class DescState:                                     # Wykorzystanie stanu deskryptora;
    w wersji 2.x należy użyć (object)

    def __init__(self, value):
        self.value = value

    def __get__(self, instance, owner):      # Przy pobieraniu atrybutów
        print('pobranie DescState')
        return self.value * 10

    def __set__(self, instance, value):      # Przy przypisywaniu atrybutów
        print('ustawienie DescState')
        self.value = value

class CalcAttrs:
    X = DescState(2)                           # Atrybut klasy deskryptora
    Y = 3                                      # Atrybut klasy

    def __init__(self):
        self.Z = 4                                # Atrybut instancji

    obj = CalcAttrs()
    print(obj.X, obj.Y, obj.Z)                  # X jest obliczane, pozostałe nie
                                                # są

    obj.X = 5                                  # Przypisanie X jest przechwytywane
                                                # Wartość przypisana ponownie
    obj.Y = 6
    atrybutowi Y w klasie
```

```

obj.Z = 7                                # Wartość przypisana ponownie
    atrybutowi Z w instancji

print(obj.x, obj.Y, obj.Z)

obj2 = CalcAttrs()                         # Atrybut X wykorzystuje
    współdzielone dane, podobnie jak Y!

print(obj2.x, obj2.Y, obj2.Z)

```

Wartość `value` znajduje się w powyższym kodzie jedynie w *deskryptorze*, dzięki czemu nie wystąpi konflikt, jeśli ta sama nazwa zostanie użyta w instancji klienta. Warto zauważyc, że zarządzamy tutaj jedynie atrybutem deskryptora — przechwytywane są próby pobrania oraz ustawienia zmiennej `X`, jednak dostęp do `Y` oraz `Z` nie (zmienna `Y` dołączona jest do klasy klienta, natomiast `Z` — do instancji). Po wykonaniu powyższego kodu zmienna `X` jest po pobraniu obliczana, ale jej wartość jest taka sama we wszystkich instancjach klienckich, ponieważ wykorzystuje ona stan deskryptora:

```

c:\code> py -3 desc-state-desc.py

pobranie DescState
20 3 4

ustawienie DescState
pobranie DescState
50 6 7

pobranie DecState
20 6 4

```

Deskryptor może także przechowywać lub wykorzystywać atrybut dołączony do instancji klasy klienta zamiast do siebie. W takim wypadku dane mogą być inne w każdej instancji klienckiej w przeciwieństwie do danych przechowywanych w samym deskryptorze. Deskryptor z poniższego przykładu zakłada, że instancja ma atrybut `_Y` dołączony do klasy klienta, i wykorzystuje go do obliczenia reprezentowanej przez niego wartości.

```

class InstState:                      # Wykorzystanie stanu instancji; w
    wersji 2.x należy użyć (object)

    def __get__(self, instance, owner):
        print('pobranie InstState')      # Założenie ustawienia przez klasę
        klienta

        return instance._Y * 100

    def __set__(self, instance, value):
        print('ustawienie InstState')
        instance._Y = value

# Klasa klienta

class CalcAttrs:
    X = DescState(2)                  # Atrybut deskryptora klasy
    Y = InstState()                   # Atrybut deskryptora klasy

    def __init__(self):

```

```

    self._Y = 3                      # Atrybut instancji
    self.Z = 4                        # Atrybut instancji
obj = CalcAttrs()
print(obj.x, obj.Y, obj.Z)          # X oraz Y są obliczane, Z nie jest
obj.x = 5                          # Przypisania X oraz Y są
przechwytywane
obj.Y = 6                          # Wartość przypisana ponownie
atrybutowi Y w klasie
obj.Z = 7                          # Wartość przypisana ponownie
atrybutowi Z w instancji
print(obj.x, obj.Y, obj.Z)          # Tym razem wartość X jest inna,
podobnie jak Z!

```

Tym razem X oraz Y przypisywane są do deskryptorów i obliczane przy pobraniu (w poprzednim przykładzie do X przypisywany był deskryptor). Nowy deskryptor z powyższego przykładu nie ma własnych informacji, jednak wykorzystuje atrybut, który zakłada, że istnieje w instancji. Atrybut ten nosi nazwę `_X` w celu uniknięcia konfliktu z nazwą z samego deskryptora. Po wykonaniu tej wersji kodu wyniki będą podobne, jednak drugi atrybut jest zarządzany — wykorzystywany jest do tego stan znajdujący się w instancji, a w nie w deskryptorze.

```

c:\code> py -3 desc-state-inst.py
pobranie InstState
20 3 4
ustawienie InstState
pobranie InstState
50 6 7
pobranie InstState
20 6 4

```

Zarówno stan z deskryptora, jak i stan z instancji pełnią swoje role. Tak naprawdę na tym właśnie polega przewaga deskryptorów nad właściwościami — ponieważ mają one własny stan, mogą z łatwością przechowywać dane wewnętrzne, bez dodawania ich do przestrzeni nazw obiektu instancji klienta. Poniższy przykład stanowi podsumowanie. Wykorzystane są w nim oba źródła informacji o stanie. Atrybut `self.data` zawiera informacje właściwe dla deskryptora, a `instance.data` informacje właściwe dla instancji klienckiej:

```

>>> class DescBoth:
        def __init__(self, data):
            self.data = data
        def __get__(self, instance, owner):
            return '%s, %s' % (self.data, instance.data)
        def __set__(self, instance, value):
            instance.data = value
>>> class Client:

```

```

def __init__(self, data):
    self.data = data
managed = DescBoth('mielonka')

>>> I = Client('jajka')
>>> I.managed           # Wyświetlenie obu źródeł danych
'mielonka, jajka'
>>> I.managed = 'MIELONKA'      # Zmiana danych instancji
>>> I.managed
'mielonka, MIELONKA'

```

Skutki wyboru miejsca przechowywania danych zostaną szerzej opisane w przykładzie w dalszej części rozdziału. Zanim przejdziemy dalej, przypomnijmy sobie z opisu slotów w rozdziale 32., że do „wirtualnych” atrybutów, takich jak właściwości i deskryptory, nawet jeżeli nie ma ich w słowniku przestrzeni nazw instancji, można odwoływać się za pomocą metod `dir` i `getattr`. Odwoływanie się do nich w ten czy inny sposób zależy od programu. W przypadku właściwości i deskryptorów można wykonywać dowolne operacje. Są to mniej oczywiste instancje „danych” niż sloty:

```

>>> I.__dict__
{'dane': 'MIELONKA'}
>>> [x for x in dir(I) if not x.startswith('__')]
['dane', 'zarządzane']
>>> getattr(I, 'dane')
'MIELONKA'
>>> getattr(I, 'zarządzane')
'mielonka, MIELONKA'
>>> for attr in (x for x in dir(I) if not x.startswith('__')):
    print('%s => %s' % (attr, getattr(I, attr)))
dane => MIELONKA
zarządzane => mielonka, MIELONKA

```

Bardziej ogólne narzędzia `__getattribute__` i `__getattro__`, opisane w dalszej części rozdziału, nie są przeznaczone do powyższych zastosowań. Ponieważ nie mają atrybutów na poziomie klasy, wyniki zwracane przez metodę `dir` nie zawierają nazwy ich „wirtualnych” atrybutów [1]. Nie są za to ograniczone do określonych nazw atrybutów zakodowanych jako właściwości lub deskryptory, które oferują znacznie więcej możliwości, o czym mowa w następnym podrozdziale.

Powiązania pomiędzy właściwościami a deskryptorami

Jak wspomniano wcześniej, właściwości i deskryptory są ze sobą silnie powiązane — wbudowana funkcja `property` jest po prostu wygodnym sposobem tworzenia deskryptora.

Skoro już wiemy, jak działają oba rozwiązania, powinniśmy być w stanie zobaczyć, że możliwe jest symulowanie funkcji wbudowanej `property` za pomocą klasy deskryptora, jak poniżej:

```
class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel                      # Zapisanie metod bez wiązania
        self.__doc__ = doc                     # lub innych obiektów
                                                # wywoływalnych

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("nie można pobrać atrybutu")
        return self.fget(instance)           # Przekazanie instancji do self
                                            # w akcesorach właściwości

    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("nie można ustawić atrybutu")
        self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("nie można usunąć atrybutu")
        self.fdel(instance)

class Person:
    def getName(self): ...
    def setName(self, value): ...
    name = Property(getName, setName)      # Użyć jak property()
x = Person()
x.name
x.name = 'Robert'
del x.name
```

Powyzsza klasa `Property` przechwytuje dostęp do atrybutów za pomocą protokołu deskryptora i przekierowuje żądania do funkcji lub metod przekazanych i zapisanych w stanie deskryptora, kiedy klasa jest tworzona. Pobranie atrybutu jest na przykład przekierowywane z klasy `Person` do metody `__get__` klasy `Property` i z powrotem do metody `getName` klasy `Person`. W przypadku deskryptorów wszystko „po prostu działa”.

```
c:\code> py -3 prop-desc-equiv.py
getName...
setName...
AttributeError: can't delete attribute
```

Warto zauważyć, że przykład odpowiednika klasy deskryptora obsługuje jedynie proste zastosowania właściwości. By skorzystać ze składni dekoratorów z @ w celu określenia operacji ustawiania i usuwania, nasza klasa Property musiałaby zostać rozszerzona za pomocą metod setter oraz deleter, zapisujących udekorowaną funkcję akcesora i zwracających obiekt właściwości (powinno tu wystarczyć self). Ponieważ funkcja wbudowana property robi to za nas, pominiemy tutaj kod takiego rozszerzenia.

Deskryptory, sloty i nie tylko

Teraz możemy sobie wyobrazić, jak można wykorzystywać deskryptory do implementowania slotów. Można zrezygnować ze słowników atrybutów instancji i tworzyć deskryptory na poziomie klasy, przechwytyjące dostęp do nazw slotów i wiążące je z przestrzenią wykorzystywaną przez instancję. Jednak w odróżnieniu od jawnego wywołania funkcji property większość slotów dzieje się w chwili automatycznego lub jawnego tworzenia klasy, gdy pojawia się atrybut `__slots__`.

Rozdział 32. zawiera więcej informacji o slotach (wraz z uzasadnieniem, dlaczego nie należy ich używać, z wyjątkiem patologicznych sytuacji).

	<p>W rozdziale 39. będziemy także wykorzystywać deskryptory do implementowania dekoratorów funkcji mających zastosowanie zarówno do funkcji, jak i metod. Jak zobaczymy, ponieważ deskryptory otrzymują zarówno instancje deskryptora, jak i klasy podmiotowej, w tej roli dobrze się sprawdzają, choć funkcje zagnieżdżone są zazwyczaj prostszym rozwiązaniem. W tym rozdziale wykorzystamy również deskryptory do przechwycenia wywołań wbudowanych metod.</p> <p>Warto również zatrzymać się na rozdziale 40., w którym dokładnie opisane jest pierwszeństwo deskryptorów danych we wspomnianym wcześniej modelu dziedziczenia klas. Jeżeli używa się metody <code>__set__</code>, wtedy deskryptory zastępują inne nazwy, przez co są wiążące i nie mogą być zastąpione nazwami znajdującymi się w słownikach instancji.</p>
---	---

Metody `__getattr__` oraz `__getattribute__`

Dotychczas omówiliśmy właściwości oraz deskryptory — narzędzia służące do zarządzania określonymi atrybutami. Metody przeciążania operatorów `__getattr__` oraz `__getattribute__` udostępniają jeszcze inne sposoby przechwytywania pobrania atrybutów dla instancji klas. Tak jak właściwości oraz deskryptory, pozwalają one wstawiać kod, który będzie wykonywany w momencie dostępu do atrybutów. Jak jednak zobaczymy, te dwie metody mogą być wykorzystywane w sposób bardziej uniwersalny. Ponieważ przechwytyują dowolne nazwy, ich rola jest znacznie szersza — umożliwiają m.in. delegowanie wywołań. W niektórych sytuacjach mogą być dodatkowo wywoływane, są jednak zbyt dynamiczne, aby mogły być umieszczone w wyniku metody `dir`.

Przechwytywanie pobierania atrybutów ma dwie odmiany, których kod tworzy się za pomocą dwóch różnych metod:

- Metoda `__getattr__` wykonywana jest dla atrybutów *niezdefiniowanych*, to znaczy atrybutów nieprzechowywanych w instancji lub dziedziczonych po jednej z jej klas.
- Metoda `__getattribute__` wykonywana jest dla *każdego* atrybutu, dlatego wykorzystując ją, trzeba uważać i unikać pętli rekurencyjnych przy przekazywaniu dostępu do atrybutów do klasy nadzędnej.

Z pierwszą z powyższych metod spotkaliśmy się w rozdziale 30. — jest ona dostępna we wszystkich wersjach Pythona. Druga dostępna jest w Pythonie 2.x dla klas w nowym stylu oraz dla wszystkich klas (domniemane: w nowym stylu) w wersji 3.x. Dwie powyższe metody są reprezentantami zbioru metod przechwytywania atrybutów, do którego należą także `__setattr__` oraz `__delattr__`. Ponieważ metody te pełnią podobne role, potraktujemy je tutaj jako jedno zagadnienie.

W przeciwieństwie do właściwości oraz deskryptorów metody te są częścią protokołu *przeciążania operatorów* Pythona — metod klas o specjalnych nazwach, dziedziczonych przez klasy podrzędne i wykonywanych automatycznie, gdy instancje użyte zostają w domniemanej wbudowanej operacji. Tak jak wszystkie metody klasy, każda z nich po wywołaniu otrzymuje pierwszy argument `self`, co daje dostęp do wszelkich wymaganych informacji o stanie instancji lub innych metodach klasy.

Metody `__getattr__` oraz `__getattribute__` są także bardziej *ogólne* od właściwości i deskryptorów — można je wykorzystywać do przechwytywania dostępu do dowolnych (nawet wszystkich) pobrań atrybutów instancji, a nie tylko określonych nazw, do których zostały one przypisane. Z tego powodu metody te są dobrze przystosowane do ogólnych wzorców kodu opartego na *delegacji* — można je wykorzystywać do implementowania obiektów opakowujących, zarządzających wszystkimi dostępami do atrybutów dla obiektów osadzonych. Z kolei w przypadku właściwości bądź deskryptorów musimy zdefiniować po jednym z nich dla każdego atrybutu, który chcemy przechwytywać. Jak się za chwilę przekonamy, w nowego rodzaju klasach rola ta jest nieco ograniczona w przypadku wbudowanych operacji, jednak obejmuje wszystkie nazwane metody opakowanego obiektu.

Wreszcie te dwie metody są o wiele *węższe* i bardziej *skoncentrowane* w swoim działaniu od wcześniej rozważanych alternatyw — przechwytyują jedynie pobrania atrybutów, a nie przypisania. By przechwytywać również zmianę atrybutu przez jego przypisanie, musimy utworzyć kod `__setattr__` — metody przeciążania atrybutów wykonywanej dla każdego ich pobrania, w przypadku której należy uważać, by uniknąć pętli rekurencyjnych poprzez przekierowanie przypisania atrybutów za pośrednictwem słownika przestrzeni nazw instancji. Choć zdarza się to o wiele rzadziej, możemy także utworzyć kod metody przeciążania operatorów `__delattr__` (która w ten sam sposób musi unikać pętli) w celu przechwycenia operacji usunięcia atrybutów. W przeciwieństwie do tego rozwiązania właściwości i deskryptory z założenia przechwytyują operacje pobierania, ustawiania *oraz* usuwania.

Większość metod przeciążania operatorów została wprowadzona we wcześniejszej części książki. Tutaj rozszerzymy jedynie informacje o ich zastosowaniach i omówimy ich rolę w szerszym kontekście.

Podstawy

Metody `__getattr__` oraz `__setattr__` wprowadzone zostały w rozdziałach 30. oraz 32.; o `__getattribute__` wspomnialiśmy krótko w rozdziale 32. W skrócie, jeśli klasa definiuje lub dziedziczy poniższe metody, zostaną one wykonane automatycznie, jeżeli instancja zostanie użyta w kontekście opisanym za pomocą komentarzy znajdujących się z prawej strony.

```
def __getattr__(self, nazwa):                      # Przy pobraniu niesdefiniowanego
    atrybutu [obiekt.nazwa]                        

def __getattribute__(self, nazwa):                  # Przy pobraniu wszystkich atrybutów
    [obiekt.nazwa]
```

```

def __setattr__(self, nazwa, wartość): # Przy przypisaniu wszystkich
    atrybutów [obiekt.nazwa=wartość]

def __delattr__(self, nazwa):           # Przy usunięciu wszystkich
    atrybutów[del obiekt.nazwa]

```

W każdej z metod `self` jest jak zwykle obiektem instancji docelowej, `nazwa` to nazwa atrybutu, do którego dostęp ma miejsce, a `wartość` to obiekt przypisywany do atrybutu. Dwie metody pobierania normalnie zwracają wartość atrybutu, natomiast pozostałe dwie nie zwracają niczego (`None`). Wszystkie metody mogą zgłaszać wyjątki oznaczające odmowę dostępu.

Przykładowo w celu przechwycenia operacji pobrania każdego atrybutu możemy użyć dowolnej z dwóch pierwszych metod wymienionych wyżej, natomiast by przechwycić przypisanie każdego atrybutu, możemy skorzystać z trzeciej. Poniższy kod wykorzystuje metodę `__getattr__` i działa poprawnie w wersjach Pythona 2.x i 3.x. W wersji 2.x nie trzeba implementować dziedziczenia klasy `object`:

```

class Catcher:

    def __getattribute__(self, name):
        print('Pobranie:', name)

    def __setattr__(self, name, value):
        print('Ustawienie:', name, value)

X = Catcher()

X.job                         # Wyświetla "Pobranie: job"
X.pay                          # Wyświetla "Pobranie: pay"
X.pay = 99                      # Wyświetla "Ustawienie: pay 99"

```

W tym konkretnym przypadku metodę `__getattribute__` można wykorzystać w taki sam sposób, ale w wersji Pythona 2.x (tylko w tej) należy zaimplementować dziedziczenie klasy `object`. Pojawia się potencjalne, trudno uchwycone niebezpieczeństwo zapętlenia kodu, o czym będzie mowa w następnym podrozdziale.

```

class Catcher(object):          # W wersji 2.x wymagane użycie
    (object)

    def __getattribute__(self, name):      # Metoda działa tak samo jak
    getattr                           # Niebezpieczeństwo zapętlenia

        print('Pobranie: %s' % name)
        ...pozostała część bez zmian...

```

Takie struktury kodu można wykorzystać do implementowania wzorca kodu *delegacji*, z którym spotkaliśmy się wcześniej, w rozdziale 31. Ponieważ wszystkie atrybuty przekierowywane są do naszych metod przechwytyujących w sposób ogólny, możemy sprawdzać ich poprawność i przekazywać je do osadzonych, zarządzanych obiektów. Poniższa klasa (zapozyczona z rozdziału 31.) śledzi na przykład każdą operację pobrania atrybutu wykonaną dla innego obiektu przekazanego do klasy opakowującej.

```

class wrapper:

    def __init__(self, object):
        self.wrapped = object           # Zapisanie obiektu

    def __getattribute__(self, attrname):

```

```

        print('Śledzenie:', attrname)           # Śledzenie pobrania
        return getattr(self.wrapped, attrname)   # Delegacja pobrania

X = Wrapper([1, 2, 3])
X.append(4)                                     # Wyświetlenie "Śledzenie:
append"

print(X.wrapped)                                # Wyświetlenie "[1, 2, 3,
4]"

```

Taka analogia nie występuje w przypadku właściwości oraz deskryptorów, z wyjątkiem pisania kodu akcesorów dla każdego możliwego atrybutu w każdym możliwym opakowanym obiekcie. Z drugiej strony, jeżeli taka ogólność nie jest wymagana, podstawowe metody dostępowe mogą być niepotrzebnie dodatkowo wywoływane podczas przypisywania wartości atrybutom. Jest to kompromis opisany w rozdziale 30. i wspomniany w przykładzie opisanym na końcu tego rozdziału.

Unikanie pętli w metodach przechwytyjących atrybuty

Metody te są stosunkowo proste w użyciu. Jedynym skomplikowanym elementem jest w nich potencjalna możliwość *zapętlania* (czyli rekurencji). Ponieważ metoda `__getattr__` wywoływana jest jedynie dla atrybutów niezdefiniowanych, może swobodnie pobierać inne atrybuty w ramach swojego kodu. Ponieważ jednak metody `__getattribute__` oraz `__setattr__` wykonywane są dla wszystkich atrybutów, musimy być ostrożni przy dostępie do innych atrybutów, by uniknąć ponownego wywoływanie ich wzajemnie i uruchomienia rekurencyjnej pętli.

Przykładowo kolejne pobranie atrybutu wewnątrz kodu metody `__getattribute__` spowoduje ponowne wywołanie tej metody, a kod zapętli się, dopóki nie wyczerpie się pamięć.

```

def __getattribute__(self, name):
    x = self.other                         # PĘTLA!

```

Powyższa metoda jest bardziej narażona na zapętlanie, niż się na pierwszy rzut oka wydaje. Odwołanie do atrybutu `self` w *dowolnym* miejscu klasy zawierającej powyższą metodę skutkuje wywołaniem metody `__getattribute__` i potencjalnym zapętlaniem, w zależności od wpisanego kodu klasy. Zadaniem tej metody jest przechwytywanie wszystkich odczytów wartości atrybutu, ale należy jej używać ostrożnie, ponieważ obejmuje swoim działaniem wszystkie atrybuty. Jeżeli w niej samej zakoduje się odczytywanie atrybutu, wtedy zapętlanie jest niemal pewne. Aby temu zapobiec, należy przekierować operację pobrania za pośrednictwem klasy nadzędnej wyżej w hierarchii, zamiast przechodzić do wersji z klasy z tego poziomu — klasa `object` zawsze będzie klasą nadzczną i świetnie się sprawdza w tej roli:

```

def __getattribute__(self, name):
    x = object.__getattribute__(self, 'other') # Wymuszenie klasy wyżej w
                                                celu uniknięcia siebie

```

W przypadku metody `__setattr__` sytuacja jest podobna (krótko zostało to opisane w rozdziale 30.). Przypisanie dowolnego atrybutu wewnątrz tej metody wywołuje ponownie `__setattr__` i tworzy podobną pętlę.

```

def __setattr__(self, name, value):
    self.other = value                      # PĘTLA!

```

Tutaj sytuacja jest podobna. Przypisanie wartości atrybutowi `self` w *dowolnym* miejscu klasy zawierającej powyższą metodę skutkuje wywołaniem metody `__setattr__`. Jednak prawdopodobieństwo zapętlenia jest znacznie większe, jeżeli przypisanie ma miejsce w samej metodzie `__setattr__`. By obejść ten problem, należy zamiast tego przypisać atrybut jako klucz

do słownika przestrzeni nazw instancji `__dict__`. Pozwala to uniknąć bezpośredniego przypisania atrybutu.

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value # Użycie słownika atrybutów w celu uniknięcia siebie
```

Choć jest to rzadziej stosowane rozwiązanie, metoda `__setattr__` może także przekazywać własne przypisania atrybutów do klasy nadzędnej wyżej w hierarchii w celu uniknięcia pętli, tak samo jak metoda `__getattribute__` (zgodnie z uwagą niżej jest to preferowane rozwiązanie w niektórych sytuacjach):

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value) # Wymuszenie wyższej klasy nadzędnej w celu uniknięcia siebie
```

Inaczej sytuacja przedstawia się w przypadku metody `__getattribute__` — w celu uniknięcia pętli *nie możemy* użyć sztuczki ze słownikiem `__dict__`.

```
def __getattribute__(self, name):
    x = self.__dict__['other'] # PĘTLA!
```

Pobranie samego atrybutu `__dict__` powoduje ponowne wywołanie metody `__getattribute__`, co wywołuje pętlę rekurencyjną. Dziwne, ale prawdziwe!

Metoda `__delattr__` w praktyce wykorzystywana jest rzadko, jeśli jednak tak jest, wywoływana jest dla każdego usunięcia atrybutu (tak samo jak `__setattr__` wywoływana jest dla każdego przypisania atrybutu). Tym samym należy uważać, by unikać pętli przy usuwaniu atrybutów, używając do tego tych samych technik — słowników przestrzeni nazw lub wywołań metod z klasy nadzędnej.



Jak wspomniano w rozdziale 30., nazwy atrybutów zaimplementowanych w klasach nowego stylu, takich jak sloty i właściwości, nie są przechowywane w słowniku `__dict__` instancji (sloty mogą nawet całkowicie wykluczyć ich istnienie). Z tego powodu, jeżeli w kodzie trzeba przypisywać wartości takim atrybutom, należy stosować metodę `object.__setattr__` w opisany wyżej sposób, a nie słownik `self.__dict__` z indeksem. To drugie podejście jest odpowiednie w przypadku klas, w których instancjach są przechowywane dane. W podstawowych narzędziach należy jednak używać identyfikatora `object`.

Pierwszy przykład

Wszystko to nie jest wcale aż tak skomplikowane, jak mogłoby wynikać z powyższych informacji. By zobaczyć, jak wykorzystać te koncepcje w praktyce, poniżej zamieszczamy ten sam pierwszy przykład, który wykorzystaliśmy dla właściwości i deskryptorów — tym razem zaimplementowany za pomocą metod przeciążania operatorów atrybutów. Ponieważ metody te są tak ogólne, sprawdzamy tutaj nazwy atrybutów, by wiedzieć, że odbywa się dostęp do atrybutu zarządzanego. Pozostałe mogą być przekazywane normalnie.

```
class Person: # Uniwersalne dla wersji 2.x i 3.x
    def __init__(self, name): # Dla [Person()]
        self._name = name # Wywołuje __setattr__!
    def __getattr__(self, attr): # Dla [obiekt.niezdefiniowany]
```

```

        if attr == 'name':                      # Przechwycenie name: nie
przechowano

            print('pobieranie...')

            return self._name                  # Nie tworzy pętli: prawdziwy
atrybut

        else:                                # Pozostałe są błędami

            raise AttributeError(attr)

def __setattr__(self, attr, value):      # Dla [obiekt.dowolny = wartość]

    if attr == 'name':

        print('modyfikacja...')

        attr = '_name'                     # Ustawienie nazw wewnętrznych

        self.__dict__[attr] = value         # Tutaj unikanie pętli

def __delattr__(self, attr):             # Dla [del obiekt.dowolny]

    if attr == 'name':

        print('usunięcie...')

        attr = '_name'                   # Tutaj także unikanie pętli

        del self.__dict__[attr]          # jednak jest to o wiele rzadsze

bob = Person('Robert Zielony')          # bob ma zarządzany atrybut

print(bob.name)                         # Wykonuje __getattr__

bob.name = 'Robert A. Zielony'          # Wykonuje __setattr__

print(bob.name)

del bob.name                            # Wykonuje __delattr__

print('*'*20)

anna = Person('Anna Czerwona')          # anna także dziedziczy właściwość

print(anna.name)

#print(Person.name.__doc__)             # Tutaj brak odpowiednika

```

Warto zauważyć, że przypisanie atrybutu w konstruktorze `__init__` także wywołuje metodę `__setattr__` — metoda ta przechwytuje *wszystkie* przypisania atrybutów, nawet te wewnątrz samej klasy. Po wykonaniu kodu zwracany jest ten sam wynik, jednak tym razem jest to efekt normalnego mechanizmu przeciążania operatorów Pythona oraz naszych metod przechwytywania atrybutów:

```

c:\code> py -3 getattr-person.py
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony

```

usunięcie...

pobieranie...

Anna Czerwona

Warto również zauważyc, że w przeciwnieństwie do właściwości i deskryptorów nie istnieje bezpośredni sposób podawania tutaj dokumentacji naszego atrybutu — zarządzane atrybuty istnieją wewnątrz kodu metod przechwytyujących, a nie jako odrębne obiekty.

Metoda `__getattribute__`

By uzyskać dokładnie taki sam rezultat za pomocą metody `__getattribute__`, należy zastąpić `__getattr__` w przykładzie następującym kodem. Ponieważ przechwytuje on pobrania wszystkich atrybutów, wersja ta musi uważać na unikanie pętli, przekazując nowe operacje pobierania do klasy nadzędnej. Nie może także zakładać, że wszystkie nieznane nazwy są błędami.

```
# Należy zastąpić metodę __getattr__ za pomocą poniższej
def __getattribute__(self, attr):                      # Dla [obiekt.dowolny]
    print('pobieranie: ' + attr)
    if attr == 'name':                                 # Przechwycenie wszystkich
        nazw
            attr = '_name'                            # Odwzorowanie na nazwę
        wewnętrzna
    return object.__getattribute__(self, attr)      # Tutaj unikanie pętli
```

Po uruchomieniu powyższego kodu uzyskamy podobny wynik jak poprzednio. Różnica będzie polega jedynie na tym, że metoda `__getattribute__` zostanie wywołana więcej razy w wyniku odczytania wartości atrybutu wewnątrz metody `__setattr__` (zainicjowanego w metodzie `__init__`):

```
c:\code> py -3 getattribute-person.py
```

pobieranie: __dict__

pobieranie: name

Robert Zielony

modyfikacja...

pobieranie: __dict__

pobieranie: name

Robert A. Zielony

usunięcie..

pobieranie: __dict__

pobieranie: __dict__

pobieranie: name

Anna Czerwona

Powyższy przykład jest odpowiednikiem rozwiązania z właściwościami i deskryptorami, jednak jest nieco sztuczny i tak naprawdę nie przedstawia działania tych narzędzi w praktyce. Metody `__getattr__` oraz `__getattribute__`, ponieważ są tak bardzo ogólne, częściej wykorzystywane są w kodzie opartym na mechanizmie delegacji (zgodnie z informacjami przedstawionymi wcześniej), gdzie dostęp do atrybutów jest sprawdzany i przekierowywany do osadzonego obiektu. Tam, gdzie musimy zarządzać *pojedynczym* atrybutem, właściwości i deskryptory sprawdzą się równie dobrze lub nawet lepiej.

Obliczanie atrybutów

Tak jak wcześniej, nasz poprzedni przykład nie robi nic specjalnego poza śledzeniem operacji pobrania atrybutów. Obliczenie wartości atrybutu przy jego pobraniu nie stanowi szczególnie większego wyzwania. Tak jak w przypadku właściwości i deskryptorów, poniższy kod tworzy wirtualny atrybut X, który po pobraniu wykonuje obliczenia.

```
class AttrSquare:

    def __init__(self, start):
        self.value = start                  # Wywołuje __setattr__!

    def __getattr__(self, attr):          # Przy pobraniu niezdefiniowanego
        atrybutu
            if attr == 'X':
                return self.value ** 2      # Wartość nie jest niezdefiniowana
            else:
                raise AttributeError(attr)

    def __setattr__(self, attr, value): # Przy przypisaniu wszystkich
        atrybutów
            if attr == 'X':
                attr = 'value'
            self.__dict__[attr] = value

A = AttrSquare(3)                      # 2 instancje klasy z przeciążaniem
operatorów

B = AttrSquare(32)                     # Każda ma inne informacje o stanie

print(A.x)                            # 3 ** 2

A.x = 4                               # 4 ** 2

print(A.x)                            # 4 ** 2

print(B.x)                            # 32 ** 2
```

Wykonanie powyższego kodu powoduje zwrócenie tego samego wyniku, jaki otrzymaliśmy wcześniej za pomocą właściwości i deskryptorów, jednak mechanizm tego skryptu oparty jest na metodach przechwytywania ogólnych atrybutów.

```
c:\code> py -3 getattr-computed.py
```

9

16

Metoda `__getattribute__`

Tak jak wcześniej, ten sam rezultat możemy uzyskać za pomocą użycia metody `__getattribute__` w miejsce `__getattr__`. Poniższy kod zastępuje metodę pobrania za pomocą `__getattribute__` i zmienia metodę przypisania `__setattribute__` w taki sposób, by uniknąć pętli, wykorzystując do tego bezpośrednie wywołania metod klasy nadzędnej zamiast kluczy słownika `__dict__`.

```
class AttrSquare:

    def __init__(self, start):
        self.value = start           # Wywołuje __setattr__!
    def __getattribute__(self, attr):   # Przy pobraniu wszystkich atrybutów
        if attr == 'X':
            return self.value ** 2      # Znowu wywołuje __getattribute__!
        else:
            return object.__getattribute__(self, attr)
    def __setattr__(self, attr, value):   # Przy przypisaniu wszystkich
        atrybutów
        if attr == 'X':
            attr = 'value'
        object.__setattr__(self, attr, value)
```

Po wykonaniu powyższej wersji kodu (plik *getattribute-computed.py*) otrzymany wynik będzie ten sam. Warto zwrócić uwagę na niejawne przekierowanie, które występuje wewnątrz metod tej klasy:

- `self.value=start` wewnątrz konstruktora wywołuje metodę `__setattr__`,
- `self.value` wewnątrz metody `__getattribute__` wywołuje ponownie metodę `__getattribute__`.

Tak naprawdę metoda `__getattribute__` za każdym razem, gdy pobieramy atrybut `X`, wywoływana jest *dwa razy*. Takie zjawisko nie występuje w wersji z metodą `__getattr__`, ponieważ atrybut `value` nie jest niezdefiniowany. Jeśli szybkość kodu ma dla nas znaczenie i chcemy tego uniknąć, należy zmodyfikować metodę `__getattribute__` w taki sposób, by do pobrania `value` również wykorzystywała klasę nadzędną:

```
def __getattribute__(self, attr):
    if attr == 'X':
        return object.__getattribute__(self, 'value') ** 2
```

Oczywiście to rozwiązanie nadal powoduje wywołanie metody z klasy nadzędnej, jednak bez dodatkowego wywołania rekurencyjnego, zanim tam dotrzemy. W celu prześledzenia tego, kiedy i jak wywoływanie są te metody, warto dodać wywołania `print`.

Porównanie metod `__getattr__` oraz `__getattribute__`

W celu podsumowania różnic w kodzie metod `__getattr__` oraz `__getattribute__` poniższy przykład wykorzystuje obydwie z nich w celu zaimplementowania trzech atrybutów: `attr1` jest atrybutem klasy, `attr2` jest atrybutem instancji, natomiast `attr3` jest wirtualnym atrybutem zarządzanym, obliczanym przy pobraniu.

```
class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):          # Tylko dla niezdefiniowanych
                                         # atrybutów
        print('pobieranie: ' + attr)      # Nie attr1: dziedziczony z klasą
        if attr == 'attr3':               # Nie attr2: przechowany w
                                         # instancji
            return 3
        else:
            raise AttributeError(attr)
X = GetAttr()
print(X.attr1)
print(X.attr2)
print(X.attr3)
print('-'*40)
class GetAttribute(object):           # (object) potrzebne jedynie w
                                         # wersji 2.x
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr):    # Dla pobrań wszystkich atrybutów
                                         # Tutaj użycie klasy nadzędnej w
                                         # celu uniknięcia pętli
        print('pobieranie: ' + attr)
        if attr == 'attr3':
            return 3
        else:
            return object.__getattribute__(self, attr)
X = GetAttribute()
print(X.attr1)
print(X.attr2)
print(X.attr3)
```

Po wykonaniu wersja z metodą `__getattr__` przechwytuje jedynie dostęp do atrybutu `attr3`, ponieważ jest on niezdefiniowany. Wersja z metodą `__getattribute__` przechwytuje z kolei operacje pobierania wszystkich atrybutów i musi przekierowywać te, którymi nie zarządza, do klasy nadrzędnej w celu uniknięcia pętli.

```
c:\code> py -3 getattr-v-getattr.py
1
2
pobieranie: attr3
3
-----
pobieranie: attr1
1
pobieranie: attr2
2
pobieranie: attr3
3
```

Choć metoda `__getattribute__` może przechwytywać więcej operacji pobierania atrybutów od `__getattr__`, w praktyce często są one stosowane wymiennie — jeśli atrybuty nie są fizycznie przechowywane, obie metody dają ten sam efekt.

Porównanie technik zarządzania atrybutami

W celu podsumowania różnic w kodzie wszystkich czterech zaprezentowanych w niniejszym rozdziale technik zarządzania atrybutami przejdźmy szybko przez bardziej rozbudowany przykład z obliczaniem atrybutów z wykorzystaniem każdej z technik, który można wykonać w wersji 3.x i 2.x. Poniższa wersja wykorzystuje *właściwości* do przechwytywania i obliczania atrybutów o nazwie `square` oraz `cube`. Warto zwrócić uwagę na to, jak ich wartości bazowe zostają przechowywane w zmiennych rozpoczynających się od znaku `_`, tak by nie wchodziły one w konflikt z samymi nazwami właściwości.

```
# Dwa dynamicznie obliczane atrybuty z właściwościami
class Powers(object):                                # (object) wymagany tylko w
wersji 2.x

    def __init__(self, square, cube):
        self._square = square                         # _square to wartość bazowa
        self._cube = cube                             # square to nazwa właściwości

    def getSquare(self):
        return self._square ** 2

    def setSquare(self, value):
        self._square = value

    square = property(getSquare, setSquare)
```

```

def getCube(self):
    return self._cube ** 3
cube = property(getCube)

X = Powers(3, 4)
print(X.square)                                # 3 ** 2 = 9
print(X.cube)                                  # 4 ** 3 = 64
X.square = 5
print(X.square)                                # 5 ** 2 = 25

```

By uzyskać to samo za pomocą *deskryptorów*, definiujemy atrybuty za pomocą pełnych klas. Warto zwrócić uwagę na to, że poniższe deskryptory przechowują wartości bazowe jako stan instancji, przez co znowu muszą korzystać z początkowych znaków `_` w celu uniknięcia konfliktu z nazwami deskryptorów. Jak zobaczymy w ostatnim przykładzie rozdziału, moglibyśmy uniknąć konieczności zmiany nazwy, przechowując zamiast tego wartości bazowe w postaci stanu deskryptora. W takim przypadku jednak nie można byłoby przechowywać danych, które różniłyby się w poszczególnych instancjach.

```

# To samo, ale z wykorzystaniem deskryptorów

class DescSquare(object):
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value

class DescCube(object):
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers(object):                         # W wersji 2.x należy użyć (object)
    square = DescSquare()
    cube = DescCube()

    def __init__(self, square, cube):
        self._square = square           # "self.square = square" także
        działa,                        # ponieważ wywołuje metodę __set__
        self._cube = cube               # deskryptora!

X = Powers(3, 4)
print(X.square)                            # 3 ** 2 = 9
print(X.cube)                             # 4 ** 3 = 64
X.square = 5
print(X.square)                            # 5 ** 2 = 25

```

By uzyskać ten sam rezultat za pomocą metody przechwytywania pobierania `__getattr__`, znów musimy przechować wartości bazowe za pomocą zmiennych poprzedzonych znakiem `_`, tak by operacje dostępu do zarządzanych zmiennych pozostały niezdefiniowane i tym samym wywołyły naszą metodę. Musimy również utworzyć kod metody `__setattr__` w celu przechwycenia operacji przypisania, a także uważać na jej potencjalne zapętlenie.

```
# To samo, ale z ogólnym przechwytywaniem niezdefiniowanego atrybutu
__getattr__

class Powers:

    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('nieznany atrybut:' + name)

    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value # Wykorzystany object
        else:
            self.__dict__[name] = value

X = Powers(3, 4)
print(X.square)                      # 3 ** 2 = 9
print(X(cube))                      # 4 ** 3 = 64
X.square = 5
print(X.square)                      # 5 ** 2 = 25
```

Ostatnie rozwiązanie — kod wykorzystujący metodę `__getattribute__` — jest podobne do wersji poprzedniej. Ponieważ jednak przechwytyjemy teraz każdy atrybut, musimy przekierować pobrania wartości bazowych do klasy nadzędnej w celu uniknięcia zapętlenia. Więcej razy jest jednak wywoływana metoda `__getattribute__`.

```
# To samo, ale z ogólnym przechwytywaniem wszystkich atrybutów za pomocą
__getattribute__

class Powers(object):                  # (object) wymagany tylko w
wersji 2.x

    def __init__(self, square, cube):
        self._square = square
```

```

        self._cube = cube

def __getattribute__(self, name):
    if name == 'square':
        return object.__getattribute__(self, '_square') ** 2
    elif name == 'cube':
        return object.__getattribute__(self, '_cube') ** 3
    else:
        return object.__getattribute__(self, name)

def __setattr__(self, name, value):
    if name == 'square':
        self.__dict__['_square'] = value      # Wykorzystany __dict__
    else:
        self.__dict__[name] = value

X = Powers(3, 4)
print(X.square)                      # 3 ** 2 = 9
print(X(cube))                      # 4 ** 3 = 64
X.square = 5
print(X.square)                      # 5 ** 2 = 25

```

Jak widać, każda technika przybiera w kodzie inną formę, jednak wszystkie cztery po wykonaniu dają ten sam rezultat:

```

9
64
25

```

Więcej informacji na temat porównania tych alternatyw, a także innych opcji, znajdziesz się w bardziej realistycznym ich zastosowaniu w przykładzie ze sprawdzaniem poprawności atrybutów w podrozdziale „Przykład — sprawdzanie poprawności atrybutów”. Najpierw jednak musimy omówić pewną pułapkę związaną z dwoma z powyższych narzędzi.

Przechwytywanie atrybutów wbudowanych operacji

Jeżeli czytasz po kolej wszyskcie rozdziały tej książki, w niektórych fragmentach znajdziesz podsumowanie materiału opisanego wcześniej, głównie tego z rozdziału 32. Dla pozostałych czytelników w tym podrozdziale tytułowe zagadnienie jest opisane w kontekście tego rozdziału.

Kiedy wprowadzałem metody `__getattr__` oraz `__getattribute__`, napisałem, że przechwytyują one operacje pobierania atrybutów, odpowiednio, niezdefiniowanych i wszystkich, co sprawia, że idealnie sprawdzają się we wzorcach kodu opartych na delegacji. Choć jest to prawdą dla atrybutów o *normalnych nazwach*, ich działanie zasługuje na dodatkowe wyjaśnienie. W przypadku atrybutów o nazwach metod pobieranych w sposób niejawny przez operacje wbudowane metody te mogą w ogóle nie być wykonane. Oznacza to, że wywołania metod przeciążających operatorów nie mogą być delegowane do opakowanych obiektów, o ile klasy opakowujące w jakiś sposób same nie zdefiniują tych metod ponownie.

Przykładowo pobranie atrybutów dla metod `__str__`, `__add__` oraz `__getitem__` wykonywane w sposób niejawnny za pomocą, odpowiednio, wyświetlania, wyrażeń ze znakiem + oraz indeksowania nie jest przekierowywane do ogólnych metod przechwytywania atrybutów Pythona 3.x. A w szczególności:

- W Pythonie 3.x dla takich atrybutów nie zostanie wykonana ani metoda `__getattr__`, ani `__getattribute__`.
- W Pythonie 2.x metoda `__getattr__` jest wykonywana dla takich atrybutów, jeśli są one niezdefiniowane w klasie.
- W Pythonie 2.x metoda `__getattribute__` jest dostępna jedynie dla klas w nowym stylu i działa tak samo jak w wersji 3.0.

Innymi słowy, w klasach Pythona 3.x (oraz klasach w nowym stylu z wersji 2.x) nie ma żadnego bezpośredniego sposobu uniwersalnego przechwytywania operacji wbudowanych, takich jak wyświetlanie czy dodawanie. W Pythonie 2.x metody wywoływanie przez takie operacje są wyszukiwane w *instancjach* w czasie wykonywania, tak jak wszystkie pozostałe atrybuty. W Pythonie 3.x takie metody są zamiast tego wyszukiwane w *klasach*. Ponieważ w wersji 3.x klasy definiuje się w nowym, a w wersji 2.x domyślnie w klasycznym stylu, zrozumiałym jest, że powyższe zjawiska mają miejsce w wersji 3.x, choć mogą też występować w klasach zdefiniowanych w nowym stylu w wersji 2.x. W wersji 2.x można jednak uniknąć tych zmian, co nie jest możliwe w wersji 3.x.

Jak wspomniałem w rozdziale 32., oficjalne (choć skąpo udokumentowane) powody wprowadzenia tych zmian wynikają z modyfikacji metaklas i optymalizacji wbudowanych operacji. Niezależnie od przyczyny, zważywszy, że wszystkie atrybuty — te nazwane w zwykły sposób, ale i pozostałe — są wciąż obsługiwane przez instancję i jej metody, nie oznacza to, że delegowanie jest wykluczone w ogóle. Jest to raczej krok w kierunku optymalizacji niejawnego działania wbudowanych operacji. Ta zmiana powoduje, że wzorce kodu oparte na delegacji w Pythonie 3.x stały się bardziej skomplikowane, ponieważ nie są w stanie w sposób ogólny przechwytywać wywołań metod przeciążania operatorów i przekierowywać ich do osadzonego obiektu.

Nie jest to problem nie do przejścia — klasy opakowujące są w stanie obejść to ograniczenie, redefiniując wszystkie niezbędne metody przeciążania operatorów w samej klasie opakowującej w celu delegowania wywołań. Te dodatkowe metody można dodawać albo ręcznie, za pomocą narzędzi, albo za pomocą definicji we wspólnych klasach nadziedziczeniu po nich. Takie rozwiązanie sprawia jednak, że klasy opakowujące wymagają więcej pracy niż wcześniej, jeśli metody przeciążania operatorów są częścią interfejsu opakowanego obiektu.

Należy pamiętać, że problem ten występuje jedynie w przypadku metod `__getattr__` oraz `__getattribute__`. Ponieważ właściwości i deskryptory definiowane są jedynie dla określonych atrybutów, nie mają one tak naprawdę w ogóle zastosowania do klas opartych na delegacji. Pojedynczej właściwości czy deskryptora nie można wykorzystać do przechwytywania dowolnych atrybutów. Co więcej, klasy definiujące zarówno metody przeciążania operatorów, jak i przechwytywanie atrybutów będą działały poprawnie bez względu na typ zdefiniowanego przechwytywania atrybutów. Nasze zaniepokojenie powinny wzbudzać jedynie klasy, które nie mają zdefiniowanych metod przeciążania operatorów, ale próbują je przechwytywać w sposób uniwersalny.

Rozważmy poniższy przykładowy plik `getattr-bultins.py`, który sprawdza różne typy atrybutów i operacji wbudowanych na instancjach klas zawierających metody `__getattr__` oraz `__getattribute__`.

```
class GetAttr:  
    eggs = 88                                # Obiekt eggs przechowywany w klasie, spam  
    w instancji  
  
    def __init__(self):
```

```

        self.spam = 77

    def __len__(self):          # Tutaj len, inaczej __getattr__ wywołane
zostanie z __len__

        print('__len__: 42')
        return 42

    def __getattr__(self, attr): # Dostarczenie __str__ po żądaniu, inaczej
pusta funkcja

        print('getattr: ' + attr)
        if attr == '__str__':
            return lambda *args: '[getattr str]'

        else:
            return lambda *args: None

class GetAttribute(object):   # W wersji 2.x object jest wymagany, w 3.x
– domniemany

    eggs = 88                 # W 2.x wszystkie automatycznie są
isinstance(object)

    def __init__(self):        # Musi jednak pochodzić od niej, by uzyskać
dostęp do narzędzi klas

                                # w nowym stylu,
                                # w tym __getattribute__, niektórych
wartości domyślnych __X__

    def __len__(self):
        print('__len__: 42')
        return 42

    def __getattribute__(self, attr):
        print('getattribute: ' + attr)
        if attr == '__str__':
            return lambda *args: '[GetAttribute str]'

        else:
            return lambda *args: None

for Class in GetAttr, GetAttribute:
    print('\n' + Class.__name__.ljust(50, '='))
    X = Class()

    X.eggs                  # Atrybut klasy
    X.spam                   # Atrybut instancji
    X.other                  # Brakujący atrybut
    len(X)                   # __len__ definiowane w sposób bezpośredni

```

```

# Klasy w nowym stylu muszą obsługiwać [], +; do bezpośredniego wywołania
redefiniować

try:                                # __getitem__?
    X[0]
except:
    print('niepowodzenie []')

try:
    X + 99                         # __add__?
except:
    print('niepowodzenie +')

try:
    X()                            # __call__? (niejawne, za pomocą funkcji
    wbudowanej)
except:
    print('niepowodzenie ()')

X.__call__()                         # __call__? (jawne, nie dziedziczone)
print(X.__str__())                  # __str__? (jawne, dziedziczone po type)
print(X)                            # __str__? (niejawne, za pomocą funkcji
    wbudowanej)

```

Po wykonaniu w Pythonie 2.x metoda `__getattribute__` otrzymuje różne niejawne pobrania atrybutów dla operacji wbudowanych, ponieważ Python normalnie szuka takich atrybutów w instancjach. Z kolei metoda `__getattribute__` nie jest wykonywana dla żadnych zmiennych przeciążania operatorów, ponieważ takie zmienne są wyszukiwane jedynie w klasach zdefiniowanych w nowym stylu.

```

c:\code> py -2 getattr-builtins.py
GetAttr=====
getattr: other
__len__: 42
getattr: __getitem__
getattr: __coerce__
getattr: __add__
getattr: __call__
getattr: __call__
getattr: __str__
[Getattr str]
getattr: __str__
[Getattr str]

```

```
GetAttribute=====
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025EA1D0>
```

Warto zwrócić uwagę na to, jak metoda `__getattr__` przechwytyuje zarówno niejawne, jak i jawne pobrania `__call__` oraz `__str__` w Pythonie 2.x. Metoda `getattribute` nie jest natomiast w stanie przechwycić niejawnych pobrań żadnej z nazw atrybutów dla operacji wbudowanych.

Tak naprawdę przypadek metody `getattribute` działa w Pythonie 2.x tak samo jak w 3.x, ponieważ by korzystać z tej metody, klasy z wersji 2.x muszą być w nowym stylu przez pochodzenie od `object`. Pochodzenie klasy od `object` w powyższym kodzie jest w wersji 3.x opcjonalne, ponieważ wszystkie klasy są tam w nowym stylu.

Po wykonaniu w Pythonie 3.x wyniki dla metody `__getattr__` różnią się jednak — żadna z wykonanych w sposób niejawny metod przeciążania operatorów nie wywołuje żadnej z metod przechwytywania atrybutów, kiedy atrybuty te są pobierane za pomocą operacji wbudowanych. Python 3.x (i ogólnie klasy zakodowane w nowym stylu) przy wyszukiwaniu takich nazw pomija normalny mechanizm przeszukiwania instancji. Natomiast metody nazwane w zwykły sposób są przechwytywane tak samo jak wcześniej.

```
c:\code> py -3 getattr-builtins.py
GetAttr=====
getattr: other
__len__: 42
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattr: __call__
<__main__.GetAttr object at 0x025D17F0>
<__main__.GetAttr object at 0x025D17F0>
GetAttribute=====
getattribute: eggs
getattribute: spam
```

```
getattribute: other
__len__: 42
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025D1870>
```

Powyższe wyniki możemy prześledzić aż do wywołań `print` w skrypcie, by przekonać się, z czego wynikają:

- Próba przechwycenia `__str__` przez metodę `__getattr__` dwukrotnie kończy się niepowodzeniem w wersji 3.x — raz dla wbudowanego wyświetlania, drugi raz dla jawnego pobrania, ponieważ zachowanie domyślne dziedziczone jest po klasie (a tak naprawdę wbudowanej klasie `object`, będącej klasą nadzczną dla wszystkich).
- Próba przechwycenia `__str__` przez metodę przechwytyującą wszystko `__getattribute__` kończy się niepowodzeniem tylko raz, podczas wbudowanej operacji wyświetlania. Pobranie w sposób jawnym pomija wersję odziedziczoną.
- W obu rozwiązaniach w wersji 3.x próba przechwycenia `__call__` kończy się niepowodzeniem dla wbudowanych wyrażeń wywołania, jednak przechwycenie jest możliwe w obu metodach w przypadku pobrania w sposób jawnym. W przeciwnieństwie do `__str__` nie istnieje żadna domyślna odziedziczona metoda `__call__`, która miałaby przewagę nad `__getattr__`.
- Obie klasy przechwytują `__len__` — dlatego, że jest to metoda zdefiniowana w sposób jawnym w samych klasach. Jej nazwa nie jest w Pythonie 3.x przekierowywana ani do `__getattr__`, ani do `__getattribute__`, jeśli usuniemy metody `__len__` klas.
- Próba przechwycenia wszystkich pozostałych operacji wbudowanych w wersji 3.x kończy się niepowodzeniem w obu rozwiązaniach.

I znów, w rezultacie metody przeciążania operatorów wykonywane w sposób niejawnym przez operacje wbudowane nigdy nie są w Pythonie 3.x przekierowywane za pośrednictwem żadnej z metod przechwytywania atrybutów. Python 3.x szuka takich atrybutów w *klasach* i całkowicie pomija wyszukiwanie w instancjach.

Sprawia to, że klasy opakowujące oparte na delegacji tworzy się o wiele trudniej w kodzie napisanym w Pythonie 3.x — jeśli opakowane klasy mogą zawierać metody przeciążania operatorów, metody te muszą być raz jeszcze zdefiniowane w klasie opakowującej w celu dokonania delegacji do opakowanego obiektu. W uniwersalnych narzędziach służących do delegacji może się to wiązać z dodaniem kilku dodatkowych metod.

Oczywiście dodanie takich metod może być po części zautomatyzowane za pomocą narzędzi rozszerzających klasy za pomocą nowych metod (mogą w tym pomóc dekoratory klas oraz metaklasy omówione w dwóch kolejnych rozdziałach). Co więcej, klasa nadzonna może być w stanie zdefiniować wszystkie te dodatkowe metody raz, tak by zostały one odziedziczone w klasach opartych na delegacji. Mimo to wzorce programowania oparte na delegacji wymagają w Pythonie 3.x dodatkowej pracy.

By zobaczyć bardziej realistyczny przykład tego zjawiska wraz z obchodzącym ten problem rozwiązaniem, warto zajrzeć do przykładu z dekoratorem `Private` z kolejnego rozdziału. Zapoznamy się tam również z innymi sposobami kodowania metod operatorów wymaganych w klasach pośredniczących w wersji 3.x, m.in. z modelem *domieszkowania nadklas*. Zobaczmy

tam także, że można wstawić metodę `__getattribute__` w klasie klienta w celu zachowania oryginalnego typu, choć metoda ta nadal nie będzie wywoływana dla metod przeciążania operatorów. Wyświetlanie będzie na przykład wciąż wykonywało metodę `__str__` zdefiniowaną w takiej klasie w sposób bezpośredni, zamiast przekierowywać żądanie za pośrednictwem metody `__getattribute__`.

W kolejnym przykładzie ożywimy ponownie kod z rozdziału o klasach. Skoro wiemy już, jak działa przechwytywanie atrybutów, będziemy w stanie wyjaśnić jego dziwniejsze aspekty.

Powrót do menedżerów opartych na delegacji

Przykład programowania zorientowanego obiektywem z rozdziału 28. prezentował klasę Manager wykorzystującą osadzanie obiektów oraz delegację metod do dostosowania klasy nadzędnej do własnych potrzeb zamiast dziedziczenia. Poniżej znajduje się kod tego przykładu, z usuniętą częścią niepotrzebnego testowania.

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)
class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay) # Osadzenie obiektu Person
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)      # Przechwycenie i delegacja
    def __getattr__(self, attr):
        return getattr(self.person, attr)          # Delegacja wszystkich
                                                # pozostały atrybutów
    def __repr__(self):
        return str(self.person)                   # Ponownie musi przeciążyć
                                                # operator (w wersji 3.x)
if __name__ == '__main__':
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(anna.lastName())
    anna.giveRaise(.10)
```

```

print(anna)
tom = Manager('Tomasz Czarny', 50000)           # Manager.__init__
print(tom.lastName())                           # Manager.__getattr__ ->
Person.lastName
tom.giveRaise(.10)                                # Manager.giveRaise ->
Person.giveRaise
print(tom)                                         # Manager.__str__ ->
Person.__str__

```

Komentarze na końcu pliku pokazują, które metody wywoływane są w operacji z tego wiersza. W szczególności warto zwrócić uwagę na to, jak wywołania `lastName` w klasie `Manager` pozostają niezdefiniowane i tym samym przekierowane do ogólnej metody `__getattr__`, a stamtąd do osadzonego obiektu `Person`. Poniżej znajduje się wynik skryptu — obiekt `anna` otrzymuje podwyżkę dziesięcioprocentową z klasy `Person`, natomiast obiekt `tom` dostaje dwudziestoprocentową, ponieważ metoda `giveRaise` jest zmodyfikowana w klasie `Manager`.

```
c:\code> py -3 getattr-delegate.py
```

Czerwona

[Person: Anna Czerwona, 110000]

Czarny

[Person: Tomasz Czarny, 60000]

Zwróćmy jednak uwagę na to, co się stanie, gdy na końcu skryptu *wyświetlimy* obiekt `Manager` za pomocą wywołania `print`. Wywołana zostaje wtedy metoda `__repr__` klasy opakowującej, która wykonuje delegację do metody `__repr__` obiektu `Person`. Mając to na uwadze, spójrzmy, co stanie się, jeśli *usuniemy* metodę `Manager.__repr__` w poniższym kodzie.

```

# Usunięcie metody __str__ klasy Manager
class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay)    # Osadzenie obiektu Person
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)          # Przechwycenie i
delegacja
    def __getattr__(self, attr):
        return getattr(self.person, attr)             # Delegacja wszystkich
pozostałych atrybutów

```

Teraz w Pythonie 3.x wyświetlanie *nie* przekierowuje pobrania atrybutu za pośrednictwem ogólnej metody przechwytyjącej w przypadku obiektów klasy `Manager`. Zamiast tego domyślna metoda wyświetlania `__repr__` odziedziczona po domniemanej klasie nadzędnej `object` zostaje odszukana i wykonana (obiekt `anna` nadal wyświetlany jest poprawnie, ponieważ klasa `Person` ma jawnie zdefiniowaną metodę `__repr__`).

```
c:\code> py -3 getattr-delegate.py
```

Czerwona

[Person: Anna Czerwona, 110000]

Czarny

```
<__main__.Manager object at 0x029E7B70>
```

Co ciekawe, takie wykonanie kodu bez metody `__repr__` faktycznie wywołuje w Pythonie 2.x metodę `__getattr__`, ponieważ atrybuty przeciążające operatory są przekierowywane za pośrednictwem tej metody, a klasy nie dziedziczą działania domyślnego metody `__repr__`.

```
c:\code> py -2 getattr-delegate.py
```

Czerwona

```
[Person: Anna Czerwona, 110000]
```

Czarny

```
[Person: Tomasz Czarny, 60000]
```

Przełączanie kodu na wykorzystującą metodę `__getattribute__` w Pythonie 3.x niewiele tutaj pomoże — tak jak `__getattr__`, *nie* jest ona wykonywana dla atrybutów przeciążania operatorów przyjmowanych w sposób niejawny przez wbudowane operacje Pythona 2.x oraz 3.x.

```
# Zastąpienie __getattr__ za pomocą __getattribute__

class Manager:                                     # W wersji 2.x należy
    # W wersji 2.x należy
    # Użyć (object)

    def __init__(self, name, pay):                  # Osadzenie obiektu
        Person

        self.person = Person(name, 'manager', pay)    # Przechwycenie i
                                                       # delegacja

    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)         # Przechwycenie i
                                                       # delegacja

    def __getattribute__(self, attr):
        print('**', attr)
        if attr in ['person', 'giveRaise']:
            return object.__getattribute__(self, attr)  # Pobranie moich
                                                       # atrybutów
        else:
            return getattr(self.person, attr)          # Delegacja dla
                                                       # wszystkich pozostałych
```

Bez względu na to, której metody przechwytywania atrybutów użyjemy w Pythonie 3.x, nadal musimy ponownie zdefiniować metodę `__repr__` w klasie `Manager` (jak powyżej) w celu przechwycenia operacji wyświetlania i przekierowania ich do osadzonego obiektu `Person`.

```
C:\code> py -3 getattr-delegate.py
```

Czerwona

```
[Person: Anna Czerwona, 110000]
```

```
** lastName
```

```
** person
```

Czarny

```
** giveRaise  
** person  
<__main__.Manager object at 0x028E0590>
```

Warto zwrócić uwagę na to, że `__getattribute__` wywoływane jest tutaj *dwa* razy dla metod — raz dla nazwy metody, a drugi raz dla pobrania osadzonego obiektu `self.person`. Moglibyśmy tego uniknąć, wykorzystując nieco inny kod, ale nadal musielibyśmy ponownie zdefiniować metodę `__repr__` w celu przechwycenia operacji wyświetlanego — choć w inny sposób (`self.person` spowodowałoby niepowodzenie dla tej metody `__getattribute__`).

```
# Inny kod __getattribute__ minimalizujący dodatkowe wywołania  
  
class Manager:  
  
    def __init__(self, name, pay):  
        self.person = Person(name, 'manager', pay)  
  
    def __getattribute__(self, attr):  
        print('**', attr)  
        person = object.__getattribute__(self, 'person')  
        if attr == 'giveRaise':  
            return lambda percent: person.giveRaise(percent+.10)  
        else:  
            return getattr(person, attr)  
  
    def __repr__(self):  
        person = object.__getattribute__(self, 'person')  
        return str(person)
```

Po wykonaniu tego rozwiązania alternatywnego nasz obiekt wyświetlany jest poprawnie, jednak tylko dlatego, że dodaliśmy jawną metodę `__repr__` w klasie opakowującej — atrybut nadal nie jest przekierowywany do naszej ogólnej metody przechwytywania atrybutów.

Czerwona

```
[Person: Anna Czerwona, 110000]  
** lastName  
  
Czarny  
** giveRaise  
[Person: Tomasz Czarny, 60000]
```

Sytuacja w skrócie wygląda tak, że klasy oparte na delegacji, takie jak `Manager`, muszą w Pythonie 3.x ponownie zdefiniować niektóre metody przeciążania operatorów (takie jak `__repr__` i `__str__`) w celu przekierowania ich do osadzonych obiektów, jednak nie muszą tego robić w wersji 2.x, o ile nie korzysta się z klas w nowym stylu. Jedynym bezpośrednim rozwiązaniem wydaje się użycie metody `__getattr__` w Pythonie 2.x lub ponowne zdefiniowanie metod przeciążających klasy w klasach opakowujących Pythona 3.x.

I znów, nie jest to nieosiągalny cel — wiele klas opakowujących potrafi przewidzieć zbiór wymaganych metod przeciążania operatorów, a narzędzia oraz klasy nadrzędne mogą zautomatyzować część tego zadania. W następnym rozdziale poznamy sposoby kodowania spełniające ten wymóg. Co więcej, nie wszystkie klasy wykorzystują metody przeciążające operatory (tak naprawdę większość klas aplikacji nie powinna tego zazwyczaj robić). Jest to jednak coś, o czym należy pamiętać w przypadku modeli kodu delegacji stosowanych w Pythonie 3.x. Kiedy metody przeciążania operatorów są częścią interfejsu obiektu, klasy opakowujące muszą obsługiwać je w sposób przenośny za pomocą lokalnej redefinicji metod.

Przykład — sprawdzanie poprawności atrybutów

Na zakończenie rozdziału przejdźmy do bardziej realistycznego przykładu, wykorzystującego w kodzie wszystkie cztery rozwiązania z zakresu zarządzania atrybutami. Przykład ten definiuje obiekt `CardHolder` z czterema atrybutami, z których trzy są zarządzane. Zarządzane atrybuty sprawdzają poprawność lub przekształcają dane po pobraniu albo przechowaniu. Wszystkie cztery wersje zwracają te same wyniki dla tego samego kodu testowego, jednak implementują atrybuty na bardzo różne sposoby. Przykłady są tutaj zamieszczone głównie z myślą o samodzielnym przestudiowaniu. Choć nie będę szczegółowo omawiał ich kodu, wykorzystują one koncepcje omawiane już w niniejszym rozdziale.

Wykorzystywanie właściwości do sprawdzania poprawności

Nasz pierwszy przykład do zarządzania trzema atrybutami wykorzystuje właściwości. Jak zwykle moglibyśmy zamiast atrybutów zarządzanych zastosować proste metody, jednak właściwości są pomocne, jeśli w istniejącym kodzie wykorzystywaliśmy już atrybuty. Właściwości wykonują kod automatycznie w momencie dostępu do atrybutów, jednak skoncentrowane są na ścisłe określonym ich zbiorze. Nie można ich wykorzystywać do przechwytywania wszystkich atrybutów w sposób uniwersalny.

By zrozumieć poniższy kod, kluczowe jest zwrócenie uwagi na to, że przypisania do atrybutów wewnętrz metody konstruktora `__init__` także wywołują metodę ustawiającą właściwości. Kiedy metoda ta przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `setName`, przekształcającą wartość i przypisującą ją do atrybutu instancji o nazwie `__name`, tak by nie wchodził on w konflikt z nazwą właściwości.

Taka zmiana nazwy jest niezbędna, ponieważ właściwości wykorzystują wspólny stan instancji i nie mają własnego. Dane przechowane są w atrybucie o nazwie `__name`, a atrybut o nazwie `name` jest zawsze właściwością, nie danymi. Jak widzieliśmy w rozdziale 31., nazwy takie jak `__name` oznaczają tzw. *atrybuty pseudoprzywatne*. Gdy Python umieszcza je w przestrzeni nazw instancji, dołącza do nich nazwę klasy. Dzięki temu można odróżnić atrybuty charakterystyczne dla danej implementacji od innych atrybutów, m.in. od właściwości, które nimi zarządzają.

Podsumowując, poniższa klasa zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `remain`, który jest w pełni wirtualny i obliczany na żądanie. Na potrzeby porównania: kod przykładu oparty na właściwościach składa się z 39 wierszy (nie uwzględniając dwóch wierszy inicjujących) i wykorzystuje składnię dziedziczenia klasy `object`, wymaganą w wersji 2.x, opcjonalną w 3.x.

```
# Plik validate_properties.py
```



```

        return self.retireage - self.age      # 0 ile niewykorzystywany jeszcze
jako atrybut

remain = property(remainGet)

```

Testowanie kodu

Poniższy kod, *validate_tester.py*, testuje naszą klasę. Należy go uruchomić jednym poleceniem, podając w argumencie nazwę modułu (większość kodu testującego można również umieścić na końcu każdego pliku albo interaktywnie importować z modułu po zaimportowaniu klasy). Dla wszystkich czterech wersji tego przykładu wykorzystamy ten sam kod sprawdzający. Po wykonaniu tworzymy dwie instancje klasy z zarządzanymi atrybutami, a także pobieramy i modyfikujemy jej różne atrybuty. Operacje, które mogą się nie powieść, opakowujemy w instrukcje *try*. Kod działa tak samo w wersjach 2.x i 3.x dzięki włączeniu funkcji *print*.

```

# Plik validate_tester.py

from __future__ import print_function # 2.x

def loadclass():

    import sys, importlib

    modulename = sys.argv[1]                      # Nazwa modułu w wierszu
    poleceń

    module = importlib.import_module(modulename)   # Import modułu o podanej
    nazwie

    print('[Using: %s]' % module.CardHolder)       # Tu metoda getattr() nie
    jest potrzebna

    return module.CardHolder

def printholder(who):

    print(who.acct, who.name, who.age, who.remain, who.addr, sep=' / ')

if __name__ == '__main__':
    CardHolder = loadclass()

    bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
    print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')
    bob.name = 'Robert A. Zielony'
    bob.age = 50
    bob.acct = '23-45-67-89'
    printholder(bob)

    anna = CardHolder('5678-12-34', 'Anna Czerwona', 35, 'ul. Poziomkowa 16')
    printholder(anna)

    try:
        anna.age = 200
    except:
        print('Niepoprawny wiek dla Anny')

```

```

try:
    anna.remain = 5
except:
    print("Nie można ustawić anna.remain")
try:
    anna.acct = '1234567'
except:
    print('Niepoprawne konto dla Anny')

```

Poniżej znajdują się wyniki dla naszego kodu samosprawdzającego uzyskane w wersjach 3.x i 2.x. I znów będą one takie same dla wszystkich wersji tego przykładu, inna będzie tylko nazwa testowanej klasy. Warto prześledzić kod w celu przekonania się, w jaki sposób wywoływane są metody klasy. Konta wyświetlane są z ukryciem niektórych znaków, imiona i nazwiska przekształcane są na standaryzowany format, a czas pozostały do emerytury obliczany jest przy pobraniu za pomocą atrybutu klasy.

```

c:\code> py -3 validate_tester.py validate_properties
[Using: <class 'validate_properties.CardHolder'>]
12345*** / robert_zielony / 40 / 19.5 / ul. Poziomkowa 15
23456*** / robert_a._zielony / 50 / 9.5 / ul. Poziomkowa 15
56781*** / anna_czerwona / 35 / 24.5 / ul. Poziomkowa 16
Niepoprawny wiek dla Anny
Nie można ustawić anna.remain
Niepoprawne konto dla Anny

```

Wykorzystywanie deskryptorów do sprawdzania poprawności

Napiszmy teraz nową wersję kodu naszego przykładu z wykorzystaniem deskryptorów zamiast właściwości. Jak widzieliśmy, deskryptory są bardzo podobne do właściwości, jeśli chodzi o funkcjonalność i pełnione role. Tak naprawdę właściwości są ograniczoną formą deskryptora. Tak jak właściwości, deskryptory zaprojektowano z myślą o obsłudze określonych atrybutów, a nie uniwersalnego dostępu do atrybutów. W przeciwieństwie do właściwości deskryptory mają własny stan i są rozwiązaniem bardziej ogólnym.

Opcja 1: sprawdzanie z wykorzystaniem współdzielonego stanu deskryptora instancji

By zrozumieć poniższy kod, istotne jest zauważenie, że przypisania atrybutów wewnętrz metody konstruktora `__init__` wywołują metody `__set__` deskryptora. Kiedy metoda konstruktora przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `Name.__set__()`, przekształcającą wartość i przypisującą ją do atrybutu deskryptora o nazwie `name`.

Wreszcie klasa ta implementuje te same atrybuty co wersja poprzednia — zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala także na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `remain`, który jest w pełni wirtualny i obliczany na

żądanie. Warto zwrócić uwagę na to, w jaki sposób musimy przechwytywać operacje przypisania do zmiennej `remain` w deskryptorze i zgłaszać wyjątek. Jak dowiedzieliśmy się wcześniej, gdybyśmy tego nie zrobili, przypisanie do tego atrybutu instancji po cichu utworzyłoby atrybut instancji ukrywający deskryptor atrybutu klasy.

Na potrzeby porównania: kod oparty na deskryptorze składa się z 45 wierszy. Aby kod był kompatybilny z wersją 2.x, zostało zastosowane dziedziczenie klasy `object` (można je pominąć, jeżeli wykorzystywana jest tylko wersja 3.x; w niczym to jednak nie przeszkadza, za to kod można uruchomić w starszej wersji, jeżeli jest zainstalowana).

```
# Plik validate_descriptors1.py: sprawdzanie z wykorzystaniem współdzielonego
stanu deskryptora instancji

class CardHolder(object):
    # Wszędzie (object)
    wymagany tylko w wersji 2.x

    acctlen = 8
    # Dane klasy

    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct
        # Dane instancji
        self.name = name
        # Te także wywołują
        metodę __set__
        self.age = age
        # Zmiana nazwy __X nie
        jest konieczna:
        self.addr = addr
        # w deskryptorze
        atrybut addr nie jest zarządzany
        # Atrybut remain nie ma
        danych

    class Name(object):
        # Zmienne klasy: lokalne
        def __get__(self, instance, owner):
            dla CardHolder
                return self.name

        def __set__(self, instance, value):
            value = value.lower().replace(' ', '_')
            self.name = value

    name = Name()

    class Age(object):
        def __get__(self, instance, owner):
            return self.age
            # Użycie danych
            deskryptora

        def __set__(self, instance, value):
            if value < 0 or value > 150:
                raise ValueError('niepoprawny wiek')
            else:
```

```

        self.age = value

age = Age()

class Acct(object):
    def __get__(self, instance, owner):
        return self.acct[:-3] + '***'

    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:           # Użycie danych
instancji klasy
            raise TypeError('niepoprawny numer konta')
        else:
            self.acct = value

acct = Acct()

class Remain(object):
    def __get__(self, instance, owner):
        return instance.retireage - instance.age      # Wywołuje Age.__get__

    def __set__(self, instance, value):
        raise TypeError('nie można ustawić remain') # Inaczej zezwolilibyśmy
na ustawienie

remain = Remain()

```

Powyższy kod uruchomiony wraz z opisany wcześniejszym skryptem testującym zwraca bardzo podobne wyniki jak wcześniej. Inna jest tylko nazwa klasy w pierwszym wierszu:

```
C:\code> python validate_tester.py validate_descriptors1
...takie same wyniki jak poprzednio z wyjątkiem nazwy klasy...
```

Opcja 2: sprawdzanie z wykorzystaniem indywidualnego stanu instancji

W tym przykładzie, inaczej niż w poprzednim, opartym na właściwości, wartość atrybutu name jest przypisana do obiektu deskryptora, a nie do instancji klasy klienckiej. Choć wartość tę można zapisać zarówno w stanie instancji, jak i deskryptora, w tym drugim przypadku nie trzeba przed nazwami umieszczać znaków podkreślenia, aby uniknąć konfliktów. W klasie CardHolder atrybut name jest zawsze obiektem deskryptora, a nie danych.

Powyższe podejście ma ten mankament, że stan zapisany w samym deskryptorze stanowi dane klasy, które są *współdzielone* przez wszystkie instancje klienckie, a więc nie mogą być różne w poszczególnych instancjach. Oznacza to, że we wszystkich instancjach stan będzie taki sam. Stan deskryptora może się zmieniać tylko w poszczególnych atrybutach.

Aby przekonać się, na czym to polega, w poprzednim przykładzie opartym na deskryptorze atrybuty instancji bob są wyświetlane po utworzeniu drugiej instancji o nazwie anna. Wartości zarządzanych atrybutów tej instancji (name, age i acct) *zastępują* atrybuty poprzedniej instancji o nazwie bob, ponieważ obie instancje współdzielą ten sam deskryptor przypisany ich klasie:

```
# Plik validate_tester2.py
```

```

from __future__ import print_function # 2.x
from validate_tester import loadclass
CardHolder = loadclass()
bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
print('bob:', bob.name, bob.acct, bob.age, bob.addr)
anna = CardHolder('5678-12-34', 'Anna Czerwona', 35, 'ul. Poziomkowa 16')
print('anna:', anna.name, anna.acct, anna.age, anna.addr) # Adres jest
# inny, dane klienckie
print('bob:', bob.name, bob.acct, bob.age, bob.addr) # name, acct, age
# nadpisane?

```

Wyniki potwierdzają podejrzenia: instancja bob przekształciła się w instancję anna!

```

c:\code> py -3 validate_tester2.py validate_descriptors1
[Using: <class 'validate_descriptors1.CardHolder'>]
bob: robert_zielony 12345*** 40 ul. Poziomkowa 15
anna: anna_czerwona 56781*** 35 ul. Poziomkowa 16
bob: anna_czerwona 56781*** 35 ul. Poziomkowa 16

```

Oczywiście, stan deskryptora można wykorzystywać do zarządzania implementacją deskryptora i danymi wspólnymi dla wszystkich instancji. Powyższy kod został napisany w celu zilustrowania tej techniki. Tym mniej więcej różnią się implikacje wynikające z użycia stanu klasy i instancji.

Jednak w tej opcji atrybuty obiektu CardHolder prawdopodobnie lepiej jest przechowywać w danych instancji, a nie danych deskryptora. Stosowanie tej samej konwencji nazewnictwa `__X` pozwala uniknąć konfliktu nazw w instancji, co jest w tym przypadku ważniejsze, ponieważ klient jest osobną klasą zawierającą własne atrybuty stanu. Poniżej przedstawione są niezbędne zmiany w kodzie. Liczba wierszy jest taka sama (wciąż równa 45):

```

# Plik validate_descriptors2.py: wykorzystany stan instancji klienckiej

class CardHolder(object): # Wszędzie (object) wymagany tylko w
wersji 2.x

    acctlen = 8 # Dane klasy
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct # Dane instancji klienckiej
        self.name = name # Te także wywołują metodę __set__
        self.age = age # Zmiana nazwy __X nie jest konieczna: w
deskryptorze
        self.addr = addr # atrybut addr nie jest zarządzany
                        # Atrybut remain nie ma danych

    class Name(object):
        def __get__(self, instance, owner): # Zmienne klasy: lokalne
dla CardHolder

```

```

        return instance.__name

    def __set__(self, instance, value):
        value = value.lower().replace(' ', '_')
        instance.__name = value

name = Name()                                     # class.name i zmieniony
atrybut

class Age(object):

    def __get__(self, instance, owner):
        return instance.__age                         # Użycie danych
deskryptora

    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
        else:
            instance.__age = value

age = Age()                                       # class.age i zmieniony
atrybut

class Acct(object):

    def __get__(self, instance, owner):
        return instance.__acct[:-3] + '***'

    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:          # Użycie danych instancji
klasy
            raise TypeError('niepoprawny numer konta')
        else:
            instance.__acct = value

acct = Acct()                                     # class.acct i zmieniona
nazwa

class Remain(object):

    def __get__(self, instance, owner):
        return instance.retireage - instance.age      # Wywołuje Age.__get__

    def __set__(self, instance, value):
        raise TypeError('nie można ustawić remain') # Inaczej zezwolilibyśmy
na ustawienie

remain = Remain()

```

Teraz zgodnie z oczekiwaniami do przechowywania atrybutów name, age i acct wykorzystywane są dane instancji (instancja bob nie ulega zmianom). Inne testy dają ten sam wynik:

```
c:\code> py -3 validate_tester2.py validate_descriptors2
[Using: <class 'validate_descriptors2.CardHolder'>]
bob: robert_zielony 12345*** 40 ul. Poziomkowa 15
anna: anna_czerwona 56781*** 35 ul. Poziomkowa 16
bob: robert_zielony 12345*** 40 ul. Poziomkowa 15
c:\code> py -3 validate_tester.py validate_descriptors2
```

...takie same wyniki jak poprzednio z wyjątkiem nazwy klasy... Drobna uwaga: w tej wersji nie jest możliwy dostęp do deskryptora klasy, ponieważ w argumencie instancji jest umieszczana wartość None (należy również zwrócić uwagę na zmienioną nazwę `_Name__name` widoczną w komunikacie o błędzie pojawiającym się przy próbie odczytania wartości atrybutu):

```
>>> from validate_descriptors1 import CardHolder
>>> bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
>>> bob.name
'robert_zielony'
>>> CardHolder.name
'robert_zielony'
>>> from validate_descriptors2 import CardHolder
>>> bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
>>> bob.name
'robert_zielony'
>>> CardHolder.name
AttributeError: 'NoneType' object has no attribute '_Name__name'
```

Problem można rozwiązać, dopisując niewielki kod wyświetlający bardziej szczegółowy komunikat, ale raczej nie ma takiej potrzeby. Ponieważ program w tej wersji zapisuje dane w instancji klienckiej, deskryptory nie mają znaczenia, chyba że są powiązane z instancją kliencką (podobnie jak zwykła metoda instancji). W rzeczywistości jest to zasadnicza zmiana wprowadzona w tej wersji kodu.

Ponieważ deskryptory są klasami, stanowią bardzo użyteczne narzędzie. Jednak oferowane przez nie funkcjonalności mogą mieć znaczny wpływ na działanie programu. Dlatego jak to zwykle bywa w programowaniu obiektowym, należy rozważnie wybierać sposoby przechowywania informacji o stanie.

Wykorzystywanie metody `__getattr__` do sprawdzania poprawności

Jak widzieliśmy, metoda `__getattr__` przechytuje wszystkie niezdefiniowane atrybuty, dzięki czemu może działać w sposób bardziej ogólny od właściwości czy deskryptorów. Na potrzeby przykładu sprawdzamy atrybut `name` w celu przekonania się, kiedy atrybut zarządzany jest pobierany. Pozostałe przechowywane są fizycznie w instancji, dzięki czemu nigdy nie docierają

do metody `__getattr__`. Choć takie rozwiązanie jest bardziej uniwersalne od zastosowania właściwości czy deskryptorów, wymagana może być dodatkowa praca imitująca koncentrację innych narzędzi na określonych atrybutach. Musimy sprawdzać nazwy w czasie wykonywania, a także napisać kod metody `__setattr__` w celu przechwycenia oraz sprawdzenia poprawności operacji przypisania do atrybutów.

Tak jak w wersjach tego przykładu z właściwością i deskryptorem, kluczowe jest zauważenie, że przypisania atrybutów wewnętrz metody konstruktora `__init__` wywołują także metodę `__setattr__` klasy. Kiedy metoda konstruktora przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `__setattr__`, przekształcającą wartość i przypisującą ją do atrybutu instancji o nazwie `name`. Dzięki przechowaniu `name` w instancji upewniamy się, że przyszłe próby dostępu nie wywołają metody `__getattribute__`. Atrybut `acct` jest z kolei przechowywany jako `_acct`, tak by późniejsze próby dostępu do `acct` wywoływały metodę `__getattribute__`.

W rezultacie powyższa klasa, jak dwie poprzednie, zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala także na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `remain`, który jest w pełni wirtualny i obliczany na żądanie.

Na potrzeby porównania: to rozwiązanie składa się z 32 wierszy kodu — 7 mniej od wersji opartej na właściwościach i 13 mniej od wersji wykorzystującej deskryptory. Jasność kodu ma oczywiście większe znaczenie od jego rozmiaru, jednak dodatkowy kod może czasami wiązać się z dodatkową pracą przy programowaniu oraz utrzymywaniu. Ważniejsze są tutaj chyba pełne *role* — narządzanie uniwersalne, takie jak `__getattribute__`, może się lepiej sprawdzać w ogólnej delegacji, natomiast właściwości i deskryptory są zaprojektowane z myślą o zarządzaniu określonymi atrybutami.

Warto również zauważać, że poniższy kod powoduje *dodatkowe wywołania* przy ustawianiu atrybutów niezarządzanych (na przykład `addr`), natomiast żadne dodatkowe wywołania nie występują dla pobierania atrybutów niezarządzanych, gdyż są one zdefiniowane. Choć dla większości programów będzie się to wiązało z niewielkim wzrostem nakładu pracy, właściwości i deskryptory powodują dodatkowe wywołanie jedynie przy dostępie do atrybutów zarządzanych. Ponadto znajdują się w wyniku zwracanym przez metodę `dir` wywoływaną przez podstawowe narzędzia.

Poniżej znajduje się wersja kodu wykorzystująca metodę `__getattribute__`.

```
# Plik validate_getattr.py

class CardHolder:

    acctlen = 8                                # Dane klasy
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                        # Dane instancji
        self.name = name                         # Te także wywołują metodę __setattr__
        self.age = age                           # Zmiana nazwy __acct nie jest
                                                # konieczna: nazwa jest sprawdzana
        self.addr = addr                         # Atrybut addr nie jest zarządzany
                                                # Atrybut remain nie ma danych

    def __getattribute__(self, name):
        if name == 'acct':                      # Przy pobraniu
            niezdefiniowanego atrybutu
```

```

        return self._acct[:-3] + '***'          # Atrybuty name, age, addr
są zdefiniowane

    elif name == 'remain':
        return self.retireage - self.age         # Nie wywołuje __getattr__

    else:
        raise AttributeError(name)

def __setattr__(self, name, value):
    if name == 'name':                      # Dla wszystkich przypisań
do atrybutów

        value = value.lower().replace(' ', '_') # Atrybut addr przechowany
w sposób bezpośredni

    elif name == 'age':                     # Zmiana nazwy acct na
_acct

        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')

    elif name == 'acct':
        name = '_acct'
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('niepoprawny numer konta')

    elif name == 'remain':
        raise TypeError('nie można ustawić remain')
    self.__dict__[name] = value             # Uniknięcie zapętlenia

```

Powyższy kod uruchomiony z dowolnym skryptem testowym zwraca takie same wyniki jak poprzednio (inna jest tylko nazwa klasy):

```

c:\code> py -3 validate_tester.py validate_getattr
... taki sam wynik jak w przykładzie z właściwościami, inna jest tylko nazwa
klasy...

c:\code> py -3 validate_tester2.py validate_getattr
... taki sam wynik jak w przykładzie z deskryptorami, inna jest tylko nazwa
klasy...

```

Wykorzystywanie metody __getattribute__ do sprawdzania poprawności

Nasz ostatni wariant wykorzystuje przechwytyującą wszystko metodę `__getattribute__` w celu przechycenia operacji pobierania atrybutów i zarządzania nimi zgodnie z potrzebami. Przechwycone zostaje każde pobranie atrybutu, dlatego musimy sprawdzać nazwy atrybutów w celu wykrycia tych zarządzanych i przekierować wszystkie pozostałe do klasy nadzędnej, tak

by pobranie zostało tam przetworzone w normalny sposób. Ta wersja wykorzystuje do przechwytywania operacji przypisania tę samą metodę `__setattr__` co wariant wcześniejszy.

Poniższy kod działa bardzo podobnie do wersji z metodą `__getattribute__`, dlatego nie będę tutaj powtarzał pełnego opisu. Warto jednak zauważyć, że ponieważ *każde* pobranie atrybutu przekierowywane jest do metody `__getattribute__`, nie musimy zmieniać nazw w celu ich przechwycenia (atribut `acct` przechowany zostaje jako `acct`). Z drugiej strony, kod ten musi uwzględniać przekierowanie operacji pobrania niezarządzanych atrybutów do klasy nadzędnej w celu uniknięcia zapętlenia.

Warto również zauważyć, że poniższa wersja powoduje wykonanie dodatkowych wywołań zarówno dla ustawiania, jak i pobierania atrybutów niezarządzanych (na przykład `addr`). Jeśli szybkość działania kodu ma duże znaczenie, to rozwiążanie może być naj wolniejsze ze wszystkich. Na potrzeby porównania: poniższa wersja składa się z 32 wierszy kodu, tak samo jak poprzednia. Zastosowane jest w niej dziedziczenie klasy `object` na potrzeby kompatybilności z wersją 2.x. Metoda `__getattribute__`, podobnie jak właściwości i deskryptory, jest narzędziem typowym dla klas definiowanych w nowym stylu.

```
# Plik validate_getattribute.py

class CardHolder (object):
    acctlen = 8                                # (object) wymagany tylko w wersji 2.x
    retireage = 59.5                             # Dane klasy

    def __init__(self, acct, name, age, addr):
        self.acct = acct                         # Dane instancji
        self.name = name                          # Te także wywołują metodę __setattr__
        self.age = age                            # Zmiana nazwy __acct nie jest
                                                # konieczna: nazwa jest sprawdzana
        self.addr = addr                          # Atrybut addr nie jest zarządzany
                                                # Atrybut remain nie ma danych

    def __getattribute__(self, name):
        superget = object.__getattribute__       # Bez pętli – jeden poziom w góre
        if name == 'acct':                      # Dla wszystkich pobrań atrybutów
            return superget(self, 'acct')[:-3] + '***'
        elif name == 'remain':
            return superget(self, 'retireage') - superget(self, 'age')
        else:
            return superget(self, name)          # name, age, addr:
                                                # przechowane

    def __setattr__(self, name, value):
        if name == 'name':                      # Dla wszystkich przypisań
            atrybutów
                value = value.lower().replace(' ', '_') # Atrybut addr przechowany
                                                # w sposób bezpośredni
        elif name == 'age':
```

```
if value < 0 or value > 150:  
    raise ValueError('niepoprawny wiek')  
  
elif name == 'acct':  
    value = value.replace('-', '')  
  
    if len(value) != self.acctlen:  
        raise TypeError('niepoprawny numer konta')  
  
elif name == 'remain':  
    raise TypeError('nie można ustawić remain')  
  
self.__dict__[name] = value # Unikanie pętli,  
oryginalne nazwy
```

Skrypty wykorzystujące metody `__getattr__` i `__getattribute__` uruchomione w wersji 2.x lub 3.x z obydwoma skryptami testowymi działają tak samo jak wersje wykorzystujące właściwości i deskryptory. Wszystkie cztery drogi prowadzą do tego samego celu, mimo że skrypty różnią się strukturą, jak również niektóre ich role mogą być zbędne. By dowiedzieć się nieco więcej na temat technik zarządzania atrybutami, warto samodzielnie przestudiować i wykonać kody przykładów z tego rozdziału.

Podsumowanie rozdziału

Niniejszy rozdział omawiał różne techniki zarządzania dostępem do atrybutów w Pythonie, w tym metody przeciążania operatorów `__getattr__` oraz `__getattribute__`, właściwości klas oraz deskryptory atrybutów. Porównaliśmy te narzędzia, przedstawiliśmy różnice między nimi oraz zaprezentowaliśmy kilka przypadków użycia demonstrujących ich działanie.

W rozdziale 39. będziemy kontynuowali omawianie elementów służących do budowy narzędzi, przyglądając się *dekoratorom* — kodowi wykonywanemu automatycznie w czasie tworzenia funkcji oraz klas, a nie w momencie dostępu do atrybutów. Zanim jednak przejdziemy dalej, zapoznajmy się ze zbiorem pytań podsumowujących informacje zaprezentowane w tym rozdziale.

Sprawdź swoją wiedzę – quiz

1. Czym różnią się od siebie metody `__getattr__` oraz `__getattribute__`?
2. Czym różnią się od siebie właściwości oraz deskryptory?
3. W jaki sposób są ze sobą powiązane właściwości oraz dekoratory?
4. Jakie są główne różnice w funkcjonalności metod `__getattr__` oraz `__getattribute__` w porównaniu z funkcjonalnością właściwości oraz deskryptorów?
5. Czy całe to porównanie nie jest po prostu rodzajem kłopotni?

Sprawdź swoją wiedzę — odpowiedzi

1. Metoda `__getattr__` wykonywana jest jedynie dla operacji pobierania atrybutów *niezdefiniowanych* — to znaczy tych, które nie są obecne w instancji i nie są dziedziczone po żadnej z jej klas. Metoda `__getattribute__` jest z kolei wywoływana dla *każdej* operacji pobrania atrybutu, bez względu na to, czy atrybut ten jest zdefiniowany, czy też nie. Z tego powodu kod znajdujący się wewnątrz metody `__getattr__` może swobodnie pobierać inne atrybuty, jeśli są one zdefiniowane, natomiast `__getattribute__` musi do pobrania takich atrybutów wykorzystywać specjalny kod umożliwiający uniknięcie zapętlenia (musi przekierować operacje pobrania do klasy nadzędnej, tak by pominąć samą siebie).
2. Właściwości pełnią rolę szczegółową, natomiast deskryptory są bardziej ogólne. Właściwości definiują funkcje pobierania, ustawiania i usuwania dla określonego atrybutu. Deskryptory także udostępniają klasę z metodami przeznaczonymi dla tych operacji, jednak dają dodatkową elastyczność umożliwiającą obsługę bardziej dowolnych działań. Tak naprawdę właściwości są prostym sposobem tworzenia szczegółowego rodzaju deskryptora — takiego, który wykonuje funkcje w momencie dostępu do atrybutu. Ich kod także się od siebie różni — właściwości tworzy się za pomocą funkcji wbudowanej, natomiast deskryptory za pomocą klasy. Tym samym deskryptory mogą korzystać ze wszystkich zwykłych opcji klas wynikających z programowania zorientowanego obiektywnego, takich jak dziedziczenie. Co więcej, poza informacjami o stanie instancji deskryptory mają własny lokalny stan, dzięki czemu są w stanie unikać konfliktów między nazwami w instancji.
3. Właściwości można tworzyć za pomocą składni dekoratorów. Ponieważ funkcja wbudowana `property` przyjmuje pojedynczy argument funkcji, można ją wykorzystać bezpośrednio jako dekorator funkcji w celu zdefiniowania właściwości dostępu przez pobranie. Z uwagi na ponowne dowiązywanie nazw dekoratorów nazwa udekorowanej funkcji przypisywana jest do właściwości, której akcesor pobierania ustawiany jest na udekorowaną oryginalną funkcję (`name = property(name)`). Atrybuty `setter` oraz `deleter` właściwości pozwalają nam na dodanie akcesorów ustawiania oraz usuwania za pomocą składni dekoratorów — ustawiają one akcesor na udekorowaną funkcję i zwracają rozszerzoną właściwość.
4. Metody `__getattr__` oraz `__getattribute__` są bardziej ogólne — można je wykorzystać do przechwytywania dowolnie wielu atrybutów. Z kolei każda właściwość lub deskryptor umożliwiają przechwytywanie dostępu jedynie dla *określonego* atrybutu — nie jesteśmy w stanie przechwycić operacji pobrania każdego atrybutu za pomocą pojedynczej właściwości czy deskryptora. Z drugiej strony, właściwości oraz deskryptory z założenia obsługują nie tylko pobieranie atrybutów, ale także *przypisanie*. Metody `__getattr__` oraz `__getattribute__` obsługują jedynie pobieranie. By przechwytywać także przypisania, niezbędne jest utworzenie metody `__setattr__`. Ich implementacja również się różni — `__getattribute__` i `__getattribute__` są metodami przeciążania operatorów, podczas gdy właściwości i deskryptory są obiektami ręcznie przypisanymi do atrybutów klas. Właściwości i deskryptory pozwalają niekiedy uniknąć dodatkowych wywołań podczas przypisywania wartości niezarządzanym atrybutom, jak również są automatycznie umieszczane w wynikach metody `dir`. Jednak ich zakres działania jest węższy i nie można przy ich użyciu osiągnąć niektórych podstawowych celów. Nowe funkcjonalności pojawiające się w miarę rozwoju języka Python oferują nowe możliwości, jednak nie w pełni uwzględniają to, co było wcześniej.
5. Wcale nie. Cytując *Latający cyrk Monty Pythona*:

Kłotonia to powiązana sekwencja stwierdzeń prowadząca do konkretnej propozycji.

Wcale nie.

Ależ tak! A nie samo zaprzeczanie.

Jeśli się z panem kłóczę, muszę zająć przeciwnie stanowisko.

Ale nie może pan tylko powtarzać: „Wcale nie”.

Ależ tak.

Wcale nie!

Ależ tak!

Wcale nie! Kłótnia to proces intelektualny. Zaprzeczanie to automatyczne przecistawianie się temu, co powie druga osoba.

(krótka przerwa)

Wcale nie.

Ależ tak.

Otóz nie.

Właśnie, że tak...

[1] Jak wspomniałem w rozdziale 31., tego rodzaju dynamiczne klasy mogą również wykorzystywać metodę `_dir_` do zwracania wynikowej listy metody `dir`. Jednak podstawowe narzędzia nie powinny wykorzystywać tego opcjonalnego interfejsu.

Rozdział 39. Dekoratory

W rozdziale poświęconym zaawansowanym aspektom klas (rozdział 32.) poznaliśmy metody statyczne oraz metody klas, a także krótko przyjrzaliśmy się składni dekoratora @, którą Python oferuje w celu zadeklarowania metod tego rodzaju. Z dekoratorami funkcji spotkaliśmy się z kolei w poprzednim rozdziale (rozdział 38.), przy okazji omawiania możliwości wykorzystania funkcji wbudowanej `property` w roli dekoratora, a także w rozdziale 29., przy omawianiu pojęcia abstrakcyjnych klas nadzędnych.

Niniejszy rozdział stanowi kontynuację zagadnień, na których poprzednio zakończyliśmy omawianie dekoratorów. Tym razem zagłębimy się w mechanizmy wewnętrzne dekoratorów i omówimy bardziej zaawansowane sposoby tworzenia własnych, nowych dekoratorów. Jak zobaczymy, wiele z koncepcji omawianych we wcześniejszych rozdziałach, takich jak przechowywanie stanu, regularnie pojawia się w połączeniu z dekoratorami.

Zagadnienie to jest stosunkowo zaawansowane, a budowanie dekoratorów zazwyczaj jest bardziej interesujące dla twórców narzędzi niż dla programistów aplikacji. Mimo to, biorąc pod uwagę, że dekoratory są coraz częściej spotykane w popularnych platformach opartych na Pythonie, ich podstawowe zrozumienie może pomóc lepiej pojąć ich rolę, nawet z punktu widzenia użytkownika.

Poza omówieniem szczegółów konstruowania dekoratorów niniejszy rozdział pełni również rolę bardziej realistycznego *studium przypadku* działania Pythona. Ponieważ przykłady są tu nieco bardziej rozbudowane od innych, spotykanych dotychczas w książce, w lepszy sposób pozwalają zilustrować łączenie kodu w bardziej rozbudowane systemy oraz narzędzia. Dodatkową zaletą jest to, że większość kodu pisanego w tym rozdziale będzie można wykorzystać jako narzędzia ogólnego przeznaczenia w programach tworzonych na co dzień.

Czym jest dekorator

Dekoracja jest sposobem określania kodu zarządzającego dla funkcji oraz klas. Same dekoratory przybierają postać obiektów wywoływalnych (to znaczy funkcji), które przetwarzają inne obiekty wywoływalne. Jak widzieliśmy wcześniej w książce, dekoratory Pythona mają dwie, powiązane ze sobą odmiany, przy czym żadna nie wymaga stosowania wersji 3.x ani klas w nowym stylu:

- *Dekoratory funkcji*, wprowadzone w wersji 2.4, wykonują ponowne dowiązania nazw w momencie definicji funkcji, udostępniając warstwę logiki, która jest w stanie zarządzać funkcjami oraz metodami bądź ich późniejszymi wywołaniami.
- *Dekoratory klas*, wprowadzone w wersjach 2.6 i 3.0, wykonują ponowne dowiązania nazw w momencie definicji klasy, udostępniając warstwę logiki, która jest w stanie zarządzać klasami bądź instancjami utworzonymi za pomocą ich późniejszego wywołania.

Mówiąc w skrócie, dekoratory udostępniają sposób wstawiania *automatycznie wykonywanego kodu* na końcu instrukcji definicji funkcji oraz klas — na końcu instrukcji `def` w przypadku dekoratorów funkcji oraz na końcu instrukcji `class` w przypadku dekoratorów klas. Taki kod może pełnić wiele różnych ról, opisanych w kolejnych podrozdziałach.

Zarządzanie wywołaniami orazinstancjami

Przykładowo w typowym zastosowaniu taki wykonywany automatycznie kod można wykorzystać do rozszerzenia wywołań funkcji oraz klas. Dzieje się tak dzięki zainstalowaniu *obiektów opakowujących* (zwanych też *pośrednikami*), które zostaną wywołane później:

Pośrednicy funkcji

Dekoratory funkcji instalują obiekty opakowujące, przechwytyjące późniejsze *wywołania funkcji* i odpowiednio je przetwarzające. Zazwyczaj wywołanie jest przekazywane do oryginalnej funkcji w celu wykonania zarządzanej operacji.

Pośrednicy interfejsu

Dekoratory klas instalują obiekty opakowujące, przechwytyjące późniejsze *wywołania tworzące instancje* i odpowiednio je przetwarzające. Zazwyczaj wywołanie jest przekazywane do oryginalnej klasy w celu utworzenia zarządzanej instancji.

Dekoratory uzyskują taki efekt dzięki automatycznemu ponownemu dowiązaniu nazw funkcji oraz klas do innych obiektów wywoływalnych na końcu instrukcji `def` oraz `class`. W momencie późniejszego wywołania te obiekty mogą wykonywać zadania, takie jak śledzenie i pomiar czasu dla wywoływania funkcji czy zarządzanie dostępem do atrybutów instancji klas.

Zarządzanie funkcjami oraz klasami

Choć większość przykładów w niniejszym rozdziale poświęcona jest obiektom opakowującym przechwytyującym późniejsze wywołania funkcji oraz klas, nie jest to jedyna możliwość wykorzystania dekoratorów:

Menedżery funkcji

Dekoratory funkcji można także wykorzystać do zarządzania obiektami *funkcji* zamiast ich późniejszymi wywołaniami — na przykład w celu zarejestrowania funkcji w API. W naszym przypadku nacisk położony zostanie na częściej wykorzystywane zastosowanie w opakowywaniu wywołań.

Menedżery klas

Dekoratory klas można również wykorzystać do bezpośredniego zarządzania *obiektami klas* zamiast wywołaniami tworzącymi instancje — na przykład w celu rozszerzenia klasy za pomocą nowych metod. Ponieważ ta rola w dużej mierze pokrywa się z zastosowaniem *metaklas* dodatkowe przypadki użycia omówimy jeszcze w kolejnym rozdziale. Jak się przekonamy, oba narzędzia są uruchamiane na koniec procesu tworzenia klasy, jednak dekorator często jest rozwiązaniem mniej obciążającym system.

Innymi słowy, dekoratory funkcji można wykorzystać do zarządzania zarówno wywołaniami funkcji, jak i obiektami funkcji, natomiast dekoratory klas — do zarządzania zarówno instancjami klas, jak i samymi klasami. Dzięki zwracaniu samego udekorowanego obiektu zamiast obiektu opakowującego dekoratory stają się prostym krokiem wykonywanym po utworzeniu funkcji oraz klas.

Bez względu na pełnioną rolę dekoratory udostępniają wygodny i jawni sposób tworzenia narzędzi przydatnych zarówno w trakcie tworzenia programu, jak i w działających systemach produkcyjnych.

Wykorzystywanie i definiowanie dekoratorów

W zależności od wykonywanych przez nas zadań z dekoratorami możemy spotkać się jako ich użytkownik bądź jako osoba je udostępniająca. Jak widzieliśmy, sam Python zawiera wbudowane dekoratory o wyspecjalizowanych rolach — deklaracje metod statycznych, tworzenie właściwości i wiele innych. Dodatkowo wiele popularnych zestawów narzędzi Pythona zawiera dekoratory wykonujące takie zadania, jak zarządzanie bazą danych czy logiką interfejsu użytkownika. W takich przypadkach możemy sobie poradzić bez wiedzy o tym, w jaki sposób tworzy się kod dekoratora.

W przypadku zadań bardziej ogólnych programiści mogą samodzielnie pisać dowolny kod własnych dekoratorów. Przykładowo dekoratory funkcji można wykorzystać do rozszerzania funkcji za pomocą kodu dodającego śledzenie wywołań, testującego poprawność argumentów w trakcie debugowania, automatycznie tworzącego i zwalniającego blokady wątków czy mierzącego czas wywołań funkcji w celu optymalizacji. Wszystkie działania, jakich dodanie do wywołania funkcji możemy sobie wyobrazić, są kandydatami na własne dekoratory funkcji.

Z drugiej strony, dekoratory funkcji zaprojektowano w taki sposób, by rozszerzały one jedynie określone *wywołanie* funkcji bądź metody, a nie cały *interfejs obiektu*. Lepiej spełniają tę rolę dekoratory klas — ponieważ mogą one przechwytywać wywołania tworzące instancje, można je wykorzystać do implementowania wszelkich zadań związanych z rozszerzaniem interfejsu obiektów czy zarządzaniem. Przykładowo własne dekoratory klas mogą śledzić lub sprawdzać wszystkie referencje do atrybutów wykonywane dla obiektu. Można je także wykorzystywać do tworzenia obiektów pośredniczących (proxy), klas singletona, a także innych popularnych wzorców projektowych. Tak naprawdę niedługo zobaczymy, że wiele z dekoratorów klas jest bardzo podobnych do wzorca projektowego *delegacji*, z którym spotkaliśmy się w rozdziale 31.

Do czego służą dekoratory

Jak większość zaawansowanych narzędzi Pythona, dekoratory nigdy nie są wymagane i niezbędne — ich funkcjonalność można często zaimplementować za pomocą prostych wywołań funkcji pomocniczych lub innych technik (a na podstawowym poziomie możemy zawsze napisać kod dowiązujący ponownie zmienne w ręczny sposób; dekoratory robią to w sposób automatyczny).

Dekoratory udostępniają jawną składnię przeznaczoną do zadań tego typu, dzięki czemu intencje programisty są jasne, możliwe jest ograniczenie powtarzalności kodu, a także zapewnienie poprawnego użycia API.

- Dekoratory mają bardzo oczywistą, *jawną* składnię, co ułatwia ich dostrzeżenie w porównaniu z wywołaniami funkcji pomocniczych, które mogą być dowolnie oddalone od podmiotowych funkcji bądź klas.
- Dekoratory stosowane są *raz*, przy definicji podmiotowej funkcji bądź klasy. Nie jest konieczne dodawanie dodatkowego kodu (który być może w przyszłości będzie musiał się zmienić) do każdego wywołania klasy bądź funkcji.
- Z uwagi na dwa wcześniejsze punkty dekoratory sprawiają, że mniej prawdopodobne jest to, iż użytkownik API *zapomni* rozszerzyć funkcję bądź klasę zgodnie z wymaganiami API.

Innymi słowy, poza samym modelem funkcjonalnym dekoratory mają pewne zalety z punktu widzenia utrzymywania kodu oraz estetyki. Co więcej, jako narzędzia strukturyzujące kod w naturalny sposób powodują one jego *hermetyzację*, co ogranicza powtarzalność i ułatwia wszelkie późniejsze zmiany.

Dekoratory mają jednak również pewne potencjalne *wady* — kiedy wstawiają logikę opakowującą, mogą zmieniać typy dekorowanych obiektów oraz powodować dodatkowe wywołania. Z drugiej strony, te same zastrzeżenia można mieć do wszystkich technik dodających do obiektów logikę opakowującą.

W niniejszym rozdziale omówimy wszystkie te ograniczenia w kontekście prawdziwego kodu. Choć wybór dekoratorów jest nadal kwestią w dużej mierze subiektywną, ich zalety są na tyle

kuszące, że w świecie Pythona stają się one dobrą praktyką. By pomóc podjąć decyzję w tej sprawie, przejdziemy teraz do szczegółów.



Dekoratory a makra

Stosowanie dekoratorów można porównać do tzw. *programowania aspektowego typowego* dla innych języków, polegającego na tworzeniu kodu automatycznie uruchamianego przed wywołaniem funkcji lub po nim. Składnia dekoratorów jest bardzo podobna do *adnotacji* w Javie (niewątpliwie została stamtąd zapożyczona), niemniej model w Pythonie jest uważany za bardziej elastyczny i uniwersalny.

Niektórzy programiści porównują dekoratory do *makr*, choć jest to dość niescisłe, a nawet mylące. Makra (na przykład dyrektywa `#define` preprocesora kodu w języku C) służą do generowania kodu, tj. zastępowania tekstu kodu i rozszerzania jego fragmentów. Natomiast dekoratory w Pythonie wykonują w trakcie działania programu określone operacje, wykorzystując dowiązywanie nazwy, obiekty wywoływalne i opakowujące. Choć oba pojęcia mają wspólne obszary zastosowań, zasadniczo się od siebie różnią zasięgiem działania, implementacją i sposobem kodowania. Porównywanie dekoratora i makra jest również niewłaściwe, jak instrukcji `import` w Pythonie i dyrektywy `#include` w języku C, dlatego że pierwsze pojęcie oznacza instrukcję wykonywaną w trakcie działania programu, a drugie operację polegającą na wstawieniu tekstu do kodu źródłowego.

Oczywiście znaczenie terminu *makro* bardzo się rozmyło w miarę upływu czasu. Dla niektórych programistów oznacza ono nawet wydzieloną serię operacji wykonywanych przez procedurę, a użytkownicy innych języków widzą w nim analogię do deskryptora. Należy jednak pamiętać, że dekoratory są to *obiekty wykonywalne*, które zarządzają innymi obiektami wykonywalnymi, ale nie rozszerzają ich. Pythona należy rozumieć i używać w kategoriach jego idiomów.

Podstawy

Zacznijmy od pierwszego przyjrzenia się działaniu dekoratorów z nieco symbolicznej perspektywy. Niedługo będziemy także pisali prawdziwy, działający kod, jednak ponieważ magia dekoratorów sprowadza się do operacji automatycznego ponownego dowiązywania, istotne jest, by najpierw zrozumieć ten typ odwzorowania.

Dekoratory funkcji

Dekoratory funkcji dostępne są w Pythonie od wersji 2.4. Jak widzieliśmy wcześniej w książce, są one w dużej mierze składniowym elementem wykonującym jedną funkcję w drugiej pod koniec instrukcji `def` i dowiązującym oryginalną nazwę funkcji do wyniku.

Zastosowanie

Dekorator funkcji jest rodzajem *deklaracji w czasie wykonywania* i dotyczy funkcji, której definicja pojawi się później. Dekorator zapisywany jest w kodzie w wierszu tuż przed instrukcją `def` definiującą funkcję bądź metodę i składa się z symbolu `@`, po którym następuje referencja do *metafunkcji* — funkcji (bądź innego obiektu wywoływalnego) zarządzającej inną funkcję.

Z punktu widzenia kodu dekoratory funkcji automatycznie odwzorowują poniższą składnię:

```
@decorator
```

```
# Udekorowanie funkcji
```

```
def F(arg):  
    ...  
    F(99) # Wywołanie funkcji
```

na jej poniższy odpowiednik, w którym `decorator` jest jednoargumentowym obiektem wywoływalnym zwracającym obiekt wywoływalny o tej samej liczbie argumentów co `F`:

```
def F(arg):  
    ...  
    F = decorator(F) # Ponowne dowiązanie nazwy funkcji  
    do wyniku dekoratora  
    F(99) # Wywołuje decorator(F)(99)
```

Takie automatyczne ponowne dowiązywanie nazw działa dla dowolnej instrukcji `def` — bez względu na to, czy jest to prosta funkcja, czy też metoda wewnętrz klasy. Kiedy później wywoływana jest funkcja `F`, tak naprawdę wywoływany jest obiekt *zwracany* przez dekorator — może to być albo inny obiekt implementujący wymaganą logikę opakowującą, albo oryginalna funkcja.

Innymi słowy, dekoracja odwzorowuje pierwszy z poniższych zapisów na drugi (choć dekorator wykonywany jest tak naprawdę raz, w czasie dekoracji):

```
func(6, 7)  
decorator(func)(6, 7)
```

Takie automatyczne ponowne dowiązywanie nazw odpowiada składni metod statycznych oraz dekoracji właściwości, z którymi spotkaliśmy się wcześniej w książce:

```
class C:  
    @staticmethod  
    def meth(...): ... # meth = staticmethod(meth)  
  
class C:  
    @property  
    def name(self): ... # name = property(name)
```

W obu przypadkach nazwa metody jest na końcu instrukcji `def` ponownie dowiązywana do wyniku wbudowanego dekoratora funkcji. Późniejsze wywołanie oryginalnej nazwy wywołuje obiekt zwrócony przez dekorację. W tych konkretnych przypadkach oryginalne nazwy są ponownie dowiązywane do routera statycznej metody i deskryptora właściwości, ale sam proces jest bardziej ogólny, o czym mowa w następnym podrozdziale.

Implementacja

Sam dekorator jest *obiektem wywoływalnym zwracającym obiekt wywoływalny*. Oznacza to, że zwraca on obiekt, który będzie wywołany później, kiedy udekorowana funkcja wywoływana jest za pomocą oryginalnej nazwy. Zwracany obiekt jest albo obiektem opakowującym przechwytyującym późniejsze wywołania, albo w jakiś sposób rozszerzoną oryginalną funkcją. Tak naprawdę dekoratory mogą być dowolnymi typami obiektów wywoływalnych i mogą zwracać dowolny typ obiektu wywoływalnego — można wykorzystać dowolną kombinację funkcji oraz klas, choć niektóre z nich lepiej sprawdzają się w pewnych kontekstach.

Przykładowo w celu wejścia do protokołu dekoracji i zarządzania funkcją tuż po jej utworzeniu możemy napisać kod dekoratora o poniższej formie:

```

def decorator(F):
    # Przetworzenie funkcji F
    return F

@decorator
def func(): ... # func = decorator(func)

```

Ponieważ oryginalna udekorowana funkcja przypisywana jest z powrotem do swojej nazwy, powyższy kod dodaje po prostu krok po utworzeniu do definicji funkcji. Taką strukturę można na przykład wykorzystać do zarejestrowania funkcji w API czy przypisania atrybutów funkcji.

W bardziej typowym zastosowaniu w celu wstawienia logiki przechwytyjącej późniejsze wywołania funkcji możemy napisać kod dekoratora zwracającego obiekt inny od oryginalnej funkcji:

```

def decorator(F):
    # Zapisanie lub użycie funkcji F
    # Zwrócenie innego obiektu wywoływalnego: zagnieżdzonej instrukcji def,
    # klasy z __call__ i tak dalej
    @decorator
    def func(): ... # func = decorator(func)

```

Powyższy dekorator wywoływany jest w czasie dekoracji, a zwracany przez niego obiekt wywoływany jest przy późniejszym wywołaniu oryginalnej nazwy funkcji. Sam dekorator otrzymuje udekorowaną funkcję — zwracany obiekt wywoływalny otrzymuje wszelkie argumenty przekazane później do nazwy udekorowanej funkcji. Tak samo działa to w przypadku *metod klas* — domniemany obiekt instancji po prostu pokazuje się w pierwszym argumencie zwracanego obiektu wywoływalnego.

Oto popularny wzorzec kodu ujmujący tę kwestię z punktu widzenia szkieletu. Dekorator zwraca obiekt opakowujący, zachowujący oryginalną funkcję w zakresie zawierającym:

```

def decorator(F): # W momencie dekoracji @
    def wrapper(*args): # W momencie wywołania
        opakowanej funkcji
            # Użycie F oraz args
            # F(*args) wywołuje oryginalną funkcję
        return wrapper
    @decorator # func = decorator(func)
    def func(x, y): # func przekazywane do F
        dekoratora
        ...
    func(6, 7) # 6, 7 przekazywane do *args
    obiektu opakowującego

```

Kiedy później wywołana zostaje nazwa `func`, tak naprawdę wywoływana jest funkcja opakowująca `wrapper` zwracana przez dekorator `decorator`. Funkcja `wrapper` może wtedy wykonać oryginalną funkcję `func`, ponieważ jest ona nadal dostępna w *zakresie zawierającym*. W kodzie tego typu każda udekorowana funkcja tworzy nowy zakres zachowujący stan.

By uzyskać to samo z użyciem *klas*, możemy przeciążyć operację wywoływania i skorzystać z atrybutów instancji w miejscu zakresów zawierających.

```
class decorator:  
    def __init__(self, func):                      # W momencie dekoracji @  
        self.func = func  
    def __call__(self, *args):                      # W momencie wywołania  
        opakowanej funkcji  
        # Użycie self.func oraz args  
        # self.func(*args) wywołuje oryginalną funkcję  
  
@decorator  
def func(x, y):                                # func = decorator(func)  
    ...                                              # func przekazywane jest do  
    __init__  
    func(6, 7)                                       # 6, 7 przekazywane do *args  
    dla __call__
```

Gdy w tym przypadku wywołamy później nazwę *func*, tak naprawdę wywołamy metodę przeciążania operatora *__call__* instancji utworzonej za pomocą dekoratora *decorator*. Metoda *__call__* może następnie wywołać oryginalną funkcję *func*, ponieważ jest ona nadal dostępna w *atrybucie instancji*. W kodzie tego typu każda udekorowana funkcja tworzy nową instancję zachowującą stan.

Obsługa dekoracji metod

Jedną istotną kwestią dotyczącą powyższego kodu opartego na klasie jest to, że choć działa on dla przechwytywania prostych wywołań *funkcji*, nie do końca tak jest w przypadku zastosowania do funkcji *metod* klas.

```
class decorator:  
    def __init__(self, func):                      # func to metoda bez instancji  
        self.func = func  
    def __call__(self, *args):                      # self jest instancją dekoratora  
        # self.func(*args) nie działa!  
        # Instancja C nie znajduje się w args!  
  
class C:  
    @decorator  
    def method(self, x, y):                        # method = decorator(method)  
        ...                                         # Ponowne dowiązanie do  
        instancji dekoratora
```

W przypadku powyższego kodu udekorowana metoda zostaje ponownie dowiązana do instancji klasy dekoratora zamiast do prostej funkcji.

Problem polega na tym, że *self* w metodzie *__call__* dekoratora otrzymuje przy późniejszym wywołaniu metodę instancję dekoratora klasy, natomiast instancja klasy C nigdy nie znajdzie się w **args*. Sprawia to, iż niemożliwe staje się przekazanie wywołania do oryginalnej metody —

obiekt dekoratora zachowuje oryginalną funkcję metody, jednak nie ma instancji, którą może do niej przekazać.

By obsłużyć zarówno funkcje, jak i metody, lepiej sprawdzi się alternatywa z funkcją zagnieżdżoną:

```
def decorator(F):                                # F jest funkcją lub metodą bez
    instancji

    def wrapper(*args):                          # instancja klasy w args[0] w
        przypadku metody

        # F(*args) wykonuje funkcję lub metodę

        return wrapper

    @decorator

    def func(x, y):                            # func = decorator(func)

        ...

    func(6, 7)                                # Tak naprawdę wywołuje
    wrapper(6, 7)

    class C:
        @decorator

        def method(self, x, y):                  # method = decorator(method)
            ...

        funkcji

    X = C()

    X.method(6, 7)                            # Tak naprawdę wywołuje
    wrapper(X, 6, 7)
```

W przypadku takiego kodu obiekt opakowujący `wrapper` otrzymuje instancję klasy `C` w pierwszym argumencie, dzięki czemu może przekazać go do oryginalnej metody i uzyskać dostęp do informacji o stanie.

Powyższa alternatywa z funkcją zagnieżdżoną działa, ponieważ Python tworzy dowiązany obiekt metody i tym samym przekazuje instancję podmiotowej klasy do argumentu `self` jedynie wtedy, gdy atrybut metody odwołuje się do prostej funkcji. Gdy odwołuje się on natomiast do instancji klasy wywoływalnej, instancja tej klasy przekazywana jest do `self` w celu udostępnienia klasie wywoływalnej jej własnych informacji o stanie. W dalszej części rozdziału zobaczymy, jakie znaczenie ma ta subtelna różnica w praktyce.

Warto również zauważyć, że funkcje zagnieżdżone są chyba najprostszym sposobem obsługi dekoracji zarówno dla funkcji, jak i metod — choć nie są jedynym dostępnym rozwiązaniem. *Deskryptory* z poprzedniego rozdziału otrzymują na przykład przy wywołaniu zarówno deskryptory, jak i instancję podmiotowej klasy. Choć jest to bardziej skomplikowane, w dalszej części rozdziału zobaczymy, w jaki sposób możemy wykorzystać to narzędzie także w tym kontekście.

Dekoratory klas

Dekoratory klas okazały się tak przydatne, że model ten został w Pythonie 2.x oraz 3.x rozszerzony, tak by pozwolić na dekorację klas. Początkowo były przyjmowane z oporami, ponieważ ich role częściowo pełniły *metaklasy*. Ostatecznie jednak dekoratory klas zostały

zaakceptowane, ponieważ dzięki nim ten sam efekt można często osiągnąć w znacznie prostszy sposób.

Dekoratory klas są silnie powiązane z dekoratorami funkcji. Tak naprawdę wykorzystują one tę samą składnię i podobne wzorce kodu. Zamiast opakowywać poszczególne funkcje bądź metody, dekoratory klas są sposobami zarządzania klasami lub opakowywania wywołań tworzących instancje w dodatkową logikę, która zarządza instancjami utworzonymi z klasy bądź je rozszerza. W tym drugim przypadku zarządzają wszystkimi interfejsami obiektu.

Zastosowanie

Z punktu widzenia składni dekoratory klas pojawiają się tuż przed instrukcjami `class` (w podobny sposób jak dekoratory funkcji pojawiają się tuż przed definicjami funkcji). Zakładając, że dekorator jest funkcją jednoargumentową zwracającą obiekt wywoływalny, poniższa składnia dekoratora klasy:

```
@decorator  
class C:  
    ...  
x = C(99) # Utworzenie instancji
```

jest odpowiednikiem poniższego kodu. Klasa jest automatycznie przekazywana do funkcji dekoratora, natomiast wynik dekoratora przypisywany jest z powrotem do nazwy klasy:

```
class C:  
    ...  
C = decorator(C) # Ponowne dowiezanie nazwy klasy  
do wyniku dekoratora  
x = C(99) # Tak naprawdę wywołuje  
decorator(C)(99)
```

W rezultacie późniejsze wywołanie nazwy klasy w celu utworzenia instancji wywołuje obiekt zwrócony przez dekorator, a nie oryginalną klasę.

Implementacja

Nowe dekoratory klas tworzy się za pomocą tych samych technik co w przypadku dekoratorów funkcji, choć czasami modyfikacje odbywają się na dwóch poziomach: konstruktorów klasy i interfejsu dostępu do instancji. Ponieważ dekorator klasy jest także *obiektem wywoływalnym zwracającym obiekt wywoływalny*, większość kombinacji funkcji i klas będzie w zupełności wystarczająca.

Bez względu na sposób zapisu w kodzie wynikiem dekoratora jest to, co zostaje wykonane przy późniejszym utworzeniu instancji. Przykładowo by po prostu zarządzać klasą tuż po jej utworzeniu, należy zwrócić oryginalną klasę.

```
def decorator(C):  
    # Przetworzenie klasy C  
    return C  
  
@decorator  
class C: ... # C = decorator(C)
```

By zamiast tego wstawić warstwę opakowującą przechwytyjącą późniejsze wywołania tworzące instancje, należy zwrócić inny obiekt wywoływalny.

```

def decorator(C):
    # Zapisanie lub użycie klasy C

    # Zwrócenie innego obiektu wywoływalnego: zagnieżdzonej instrukcji def,
    # klasy z __call__ i tak dalej

    @decorator
    class C: ...                                # C = decorator(C)

Obiekt wywoływalny zwracany przez tego typu dekorator klasy zazwyczaj tworzy i zwraca nową
instancję oryginalnej klasy, rozszerzoną w jakiś sposób umożliwiający zarządzanie jej
interfejsem. Przykładowo poniższy kod wstawia obiekt przechwytyujący niezdefiniowane
atrybuty instancji klasy.

def decorator(cls):                           # W momencie dekoracji @
    class Wrapper:
        def __init__(self, *args):           # W momencie tworzenia instancji
            self.wrapped = cls(*args)
        def __getattr__(self, name):         # W momencie pobrania atrybutu
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C:                                     # C = decorator(C)

    def __init__(self, x, y):                 # Wykonywane przez
        Wrapper.__init__                     # Wykonuje Wrapper.__getattribute__,
        self.attr = 'spam'

x = C(6, 7)                                  # Tak naprawdę wywołuje
Wrapper(6, 7)

print(x.attr)                                 # Wykonuje Wrapper.__getattribute__,
wyświetla "spam"

```

W powyższym przykładzie dekorator dowiązuje ponownie nazwę klasy do innej klasy, zachowując oryginalną klasę w zakresie zawierającym i tworząc oraz osadzając instancję oryginalnej klasy przy wywołaniu. Kiedy atrybut jest później pobierany z instancji, jest on przechwytywany przez metodę `__getattribute__` obiektu opakowującego i delegowany do osadzonej instancji oryginalnej klasy. Co więcej, każda udekorowana klasa tworzy nowy zakres pamiętający oryginalną klasę. W dalszej części rozdziału przekształcimy ten przykład w bardziej przydatny kod.

Podobnie jak dekoratory funkcji, dekoratory klas są często zapisywane w kodzie jako funkcje fabryczne, tworzące i zwracające obiekty wywoływalne, lub jako klasy wykorzystujące metody `__init__` albo `__call__` do przechwytywania operacji wywoływania — bądź dowolna kombinacja obu rozwiązań. Funkcje fabryczne zazwyczaj zachowują stan w referencjach do zakresu zawierającego, natomiast klasy — w atrybutach.

Obsługa większej liczby instancji

Tak jak w przypadku dekoratorów funkcji, w dekoratorach klas pewne typy kombinacji działają lepiej od innych. Rozważmy poniższą niepoprawną alternatywę dla dekoratora klasy z poprzedniego przykładu.

```

class Decorator:

    def __init__(self, C):                      # W momencie dekoracji @
        self.C = C

    def __call__(self, *args):                   # W momencie tworzenia instancji
        self.wrapped = self.C(*args)
        return self

    def __getattr__(self, attrname):            # W momencie pobrania atrybutu
        return getattr(self.wrapped, attrname)

@Decorator
class C: ...                                # C = Decorator(C)

x = C()                                     # Nadpisuje x!
y = C()

```

Powyższy kod obsługuje kilka udekorowanych klas (każda z nich tworzy nową instancję klasy `Decorator`) i przechytuje wywołania tworzące instancje (każda wykonuje metodę `__call__`). W przeciwieństwie do poprzedniej wersji ta nie jest w stanie obsłużyć *większej liczby instancji* danej klasy — każde wywołanie tworzące instancję nadpisuje poprzednio zapisaną instancję. Wersja oryginalna obsługuje większą liczbę instancji, ponieważ każde wywołanie tworzące instancję tworzy nowy, niezależny obiekt opakowujący. Mówiąc bardziej ogólnie, każdy z poniższych wzorców obsługuje większą liczbę opakowanych instancji:

```

def decorator(C):                           # W momencie dekoracji @
    class Wrapper:
        def __init__(self, *args):          # W momencie tworzenia instancji
            self.wrapped = C(*args)

        return Wrapper

    class Wrapper: ...

    def __call__(self, *args):             # W momencie dekoracji @
        # W momencie tworzenia instancji
        return self.wrapped(*args)         # Osadzenie instancji w
instancji

    return __call__

```

Zjawisko to omówimy w bardziej realistycznym kontekście w dalszej części rozdziału. W praktyce musimy uważać na poprawne łączenie typów wywoływalnych, tak by obsługiwały one wyznaczone przez nas zadania.

Zagnieżdżanie dekoratorów

Czasami jeden dekorator nie wystarczy. Założymy na przykład, że przygotowaliśmy dwa dekoratory, które będą nam potrzebne podczas kodowania: jeden do sprawdzania typów argumentów przed wywołaniem funkcji, a drugi do sprawdzania typu zwracanego wyniku po wywołaniu funkcji. Każdego z dekoratorów można używać osobno, ale co robić, gdy *oba* trzeba przypisać do tej samej funkcji? Otóż należy je *zagnieździć* tak, aby wynik pierwszego

dekoratora był modyfikowany przez drugi. Kolejność zagnieźdżenia nie jest istotna, o ile tylko oba dekoratory są wykonywane podczas późniejszych wywołań.

By obsłużyć kilka kroków rozszerzenia, składnia dekoratorów pozwala na dodawanie większej liczby warstw logiki opakowującej do udekorowanej funkcji lub metody. Przy skorzystaniu z tej możliwości każdy dekorator musi się pojawiać w osobnym wierszu. Następująca postać składni dekoratorów:

```
@A  
@B  
@C  
def f(...):  
    ...
```

wykonuje to samo co poniższy kod:

```
def f(...):  
    ...  
    f = A(B(C(f)))
```

W kodzie oryginalna funkcja przekazywana jest przez trzy różne dekoratory, a wynikowy obiekt wywoływalny przypisywany jest z powrotem do oryginalnej nazwy. Każdy dekorator przetwarza wynik poprzedniego, którym może być oryginalna funkcja lub wstawiony obiekt opakowujący.

Jeśli każdy z dekoratorów wstawia obiekty opakowujące, rezultat będzie taki, że przy wywołaniu nazwy oryginalnej funkcji wywołane zostaną trzy różne warstwy logiki obiektów opakowujących, pozwalające na rozszerzenie oryginalnej funkcji na trzy różne sposoby. Ostatni wymieniony dekorator zostanie zastosowany jako pierwszy — jako zagnieźdzony najgłębiej.

Tak samo jak w przypadku funkcji, większa liczba dekoratorów klas daje większą liczbę wywołań zagnieżdzonych funkcji i być może większą liczbę poziomów logiki opakowującej wokół wywołań tworzących instancje. Przykładowo poniższy kod:

```
@spam  
@eggs  
class C:  
    ...  
X = C()
```

jest odpowiednikiem następującego:

```
class C:  
    ...  
C = spam(eggs(C))  
X = C()
```

I znów, każdy z dekoratorów może zwracać albo oryginalną klasę, albo wstawiony obiekt opakowujący. W przypadku obiektów opakowujących, gdy zostanie zażądana instancja oryginalnej klasy C, wywołanie przekierowywane jest do obiektów warstw opakowujących udostępnianych przez dekoratory spam oraz eggs, które mogą pełnić zupełnie różne role. Na przykład mogą rejestrować działanie funkcji i kontrolować dostęp do atrybutów, przy czym oba kroki będą wykonywane przy następnych żądaniach.

Przykładowo poniższe nic nierobiące dekoratory po prostu zwracają udekorowaną funkcję.

```
def d1(F): return F
def d2(F): return F
def d3(F): return F

@d1
@d2
@d3

def func():                      # func = d1(d2(d3(func)))
    print('mielonka')
func()                           # Wyświetla "mielonka"
```

Ta sama składnia działa w przypadku klas, podobnie jak te same nic nierobiące dekoratory.

Kiedy dekoratory wstawiają obiekty funkcji opakowujących, mogą one jednak rozszerzać oryginalne funkcje przy wywołaniu. Poniższy kod dokonuje konkatenacji wyniku w warstwach dekoratorów, w miarę przechodzenia warstw od najbardziej wewnętrznej do zewnętrznej.

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3

def func():                      # func = d1(d2(d3(func)))
    return 'mielonka'
print(func())                    # Wyświetla "XYZmielonka"
```

Funkcje lambda wykorzystane zostają tutaj do zaimplementowania warstw opakowujących (każda zachowuje opakowaną funkcję w zakresie zawierającym). W praktyce warstwy opakowujące mogą przybierać postać na przykład funkcji czy klas wywoływalnych. Przy dobrym zaprojektowaniu zagnieżdżanie pozwala łączyć ze sobą poszczególne etapy rozszerzania na wiele sposobów.

Argumenty dekoratorów

Dekoratory zarówno funkcji, jak i klas zdają się móc przyjmować *argumenty*, choć tak naprawdę argumenty te są przekazywane do obiektu wywoływalnego *zwracającego* z kolei dekorator, który natomiast zwraca obiekt wywoływalny. Zazwyczaj w takim przypadku definiowanych jest kilka poziomów zachowywania stanu. Poniższy kod:

```
@decorator(A, B)
def F(arg):
    ...
F(99)
```

automatycznie odwzorowywany jest na następującą postać równoważną, gdzie decorator jest obiektem wywoływalnym zwracającym sam dekorator. Zwrócony dekorator zwraca z kolei obiekt wywoływalny wykonywany później dla wywołań nazwy oryginalnej funkcji.

```
def F(arg):  
    ...  
  
    F = decorator(A, B)(F)      # Ponowne dowiązanie F do wyniku wartości  
    zwracanej przez dekorator  
  
    F(99)                      # Tak naprawdę wywołuje decorator(A, B)(F)(99)
```

Argumenty dekoratorów są ustalane przed wystąpieniem dekoracji i zazwyczaj wykorzystywane są do przechowania informacji o stanie do użycia w późniejszych wywołaniach. Funkcja dekoratora z poniższego przykładu może na przykład przybrać następującą postać:

```
def decorator(A, B):  
    # Zapisanie lub użycie A, B  
  
    def actualDecorator(F):  
        # Zapisanie lub użycie funkcji F  
  
        # Zwrócenie obiektu wywoływalnego: zagnieżdzonej instrukcji def, klasy  
        # z __call__ i tak dalej  
        return callable  
  
    return actualDecorator
```

W tej strukturze funkcja zewnętrzna zapisuje argumenty dekoratora w postaci informacji o stanie, by były one dostępne do użycia w samym dekoratorze, zwracanym przez niego obiekcie wywoływalnym lub w obu tych miejscach. Powyższy fragment kodu zachowuje argument informacji o stanie w referencjach do zakresu funkcji zawierającej, jednak często wykorzystywane są również atrybuty klas.

Innymi słowy, argumenty dekoratora często wymuszają *trzy poziomy obiektów wywoływalnych*: obiekt przyjmujący argumenty dekoratora zwracający obiekt służący jako dekorator, który z kolei zwraca obiekt obsługujący wywołania do oryginalnej funkcji bądź klasy. Każdy z tych trzech poziomów może być funkcją lub klasą i może zachowywać stan w postaci zakresów lub atrybutów klas.

Argumenty dekoratorów można wykorzystywać do przekazywania wartości inicjujących atrybuty, etykiet komunikatów, nazw weryfikowanych atrybutów itp. Można w nich umieszczać wszelkiego rodzaju parametry konfiguracyjne obiektów lub ich pośredników. Konkretnie przykłady użycia argumentów dekoratorów zobaczymy w dalszej części rozdziału.

Dekoratory zarządzają także funkcjami oraz klasami

Choć większa część reszty niniejszego rozdziału skupia się na opakowywaniu późniejszych wywołań funkcji oraz klas, powinienem podkreślić, że mechanizm dekoratorów jest o wiele bardziej uniwersalny — jest to protokół służący do przekazywania funkcji oraz klas za pośrednictwem obiektów wywoływalnych natychmiast po ich utworzeniu. Tym samym można go również wykorzystać do wywołania dowolnego przetwarzania po utworzeniu.

```
def decorate():  
    # Zapisanie lub rozszerzenie funkcji bądź klasy 0  
  
    return 0
```

```

@decorator
def F(): ...                                # F = decorator(F)

@decorator
class C: ...                                # C = decorator(C)

```

Dopóki będziemy zwracać w ten sposób oryginalny udekorowany obiekt, a nie obiekt opakowujący, możemy zarządzać samymi funkcjami i klasami, a nie tylko późniejszymi ich wywołaniami. Bardziej realistyczne przykłady takiego działania zobaczymy w dalszej części rozdziału, gdzie technika ta posłuży do rejestrowania obiektów wywoływalnych w API za pomocą dekoracji oraz przypisywania atrybutów do funkcji w czasie ich tworzenia.

Kod dekoratorów funkcji

Przejdźmy do kodu. W dalszej części niniejszego rozdziału będziemy omawiali działające przykłady demonstrujące zaprezentowane właśnie kwestie związane z dekoratorami. W niniejszym podrozdziale zaprezentujemy kilka przykładów działania dekoratorów funkcji, natomiast w kolejnym — działanie dekoratorów klas. Na koniec zamknijmy rozdział kilkoma większymi studiami przypadku użycia dekoratorów klas oraz funkcji — pełną implementacją prywatności klas i sprawdzaniem zakresów wartości argumentów.

Śledzenie wywołań

Na początek wróćmy do przykładu śledzenia wywołań z rozdziału 32. Poniższy kod definiuje oraz stosuje dekorator funkcji zliczający liczbę wywołań wykonywanych do udekorowanej funkcji, a także wyświetlający komunikat śledzenia dla każdego wywołania.

```

# Plik decorator1.py

class tracer:

    def __init__(self, func):      # W momencie dekoracji @: zapisanie
        self.calls = 0
        self.func = func
    def __call__(self, *args):      # W momencie późniejszego wywołania:
        self.calls += 1
        print('wywołanie %s %s' % (self.calls, self.func.__name__))
        self.func(*args)

@tracer
def spam(a, b, c):              # spam = tracer(spam)
    print(a + b + c)            # Opakowuje spam w obiekt dekoratora

```

Warto przyjrzeć się temu, jak każda funkcja udekorowana za pomocą tej klasy będzie tworzyła nową instancję, z własnym zapisanym obiektem funkcji oraz licznikiem wywołań. Na uwagę zasługuje także to, w jaki sposób składnia argumentów `*args` wykorzystywana jest do spakowania i rozpakowania dowolnej liczby przekazanych argumentów. Taka uniwersalność

pozwala na wykorzystanie dekoratora do opakowania dowolnej funkcji z dowolną liczbą argumentów (powyższa wersja nie działa jeszcze na metodach klas, ale w dalszej części podrozdziału to poprawimy).

Jeśli teraz zimportujemy funkcję z tego modułu i przetestujemy ją w sesji interaktywnej, otrzymamy następujące działanie. Każde wywołanie generuje początkowo komunikat śledzenia, ponieważ przechwytywane jest przez klasę dekoratora. Poniższy kod może zostać wykonany w Pythonie 2.x oraz 3.x, podobnie jak pozostałe przykłady z rozdziału, o ile nie zostało zaznaczone inaczej. (W kodzie zostały użyte instrukcje print niezależne od wersji języka, a dekoratory nie wymagają tworzenia klas w nowym stylu. Ponadto niektóre adresy szesnastkowe zostały skrócone).

```
>>> from decorator1 import spam

>>> spam(1, 2, 3)                                # Tak naprawdę wywołuje opakowujący
                                                obiekt śledzenia
                                                wywołanie 1 spam
                                                6
>>> spam('a', 'b', 'c')                          # Wywołuje metodę __call__ w klasie
                                                wywołanie 2 spam
                                                abc
>>> spam.calls                                 # Zliczenie wywołań w informacjach o
                                                stanie obiektu opakowującego
                                                2
>>> spam
<decorator1.tracer object at 0x02D9A730>
```

Po wykonaniu klasa tracer zapisuje udekorowaną funkcję i przechwytuje jej późniejsze wywołania w celu dodania warstwy logiki zliczającej i wyświetlającej każde wywołanie. Warto zwrócić uwagę na pokazywanie się całkowitej liczby wywołań jako atrybutu udekorowanej funkcji — spam jest tak naprawdę instancją klasy tracer po udekorowaniu (dekoratory mogą kopować oryginalną wartość atrybutu __name__, jednak ten proceder ma pewne ograniczenia i może być przyczyną zamieszania).

W przypadku wywołań funkcji składnia dekoracji ze znakiem @ może być wygodniejsza od modyfikowania każdego wywołania w celu uzyskania dodatkowego poziomu logiki, gdyż pozwala uniknąć przypadkowego bezpośredniego wywołania oryginalnej funkcji. Rozważmy odpowiednik tego kodu niezawierający dekoratora, jak poniższy:

```
calls = 0

def tracer(func, *args):
    global calls
    calls += 1
    print('wywołanie %s %s' % (calls, func.__name__))
    func(*args)

def spam(a, b, c):
    print(a, b, c)
```

```
>>> spam(1, 2, 3)                                # Normalne, nieśledzone wywołanie:  
przypadkowe?  
1 2 3  
  
>>> tracer(spam, 1, 2, 3)                      # Specjalne śledzone wywołanie bez  
dekoratorów  
wywołanie 1 spam  
1 2 3
```

Powyższą alternatywę można wykorzystać w dowolnej funkcji bez specjalnej składni ze znakiem @, jednak w przeciwieństwie do wersji z dekoratorem wymaga ona dodania dodatkowej składni w każdym miejscu, w którym w kodzie wywoływana jest funkcja. Co więcej, jej cel nie jest tak oczywisty i brak jest także zapewnienia, że dodatkowa warstwa zostanie wywołana dla normalnych wywołań. Choć dekoratory nigdy nie są *wymagane* (nazwy zawsze możemy dowieźć ponownie ręcznie), często są najwygodniejszą opcją.

Możliwości w zakresie zachowania informacji o stanie

Ostatni przykład zwraca uwagę na pewną istotną kwestię. Dekoratory funkcji mają kilka możliwości w zakresie zachowywania informacji o stanie udostępnianych w czasie dekoracji i wykorzystywanych w czasie samego wywołania funkcji. Zazwyczaj muszą one obsługiwać większą liczbę udekorowanych obiektów oraz wywołań, jednak istnieje kilka sposobów implementacji tych celów — do zachowania stanu można wykorzystać atrybuty instancji, zmienne globalne, zmienne nielokalne, a także atrybuty funkcji.

Atrybuty instancji klasy

Przykładowo poniżej znajduje się rozszerzona wersja poprzedniego przykładu, dodająca obsługę argumentów ze słowami kluczowymi i operatorem **, zwracającą wynik opakowanej funkcji, tak by możliwa była obsługa większej liczby zastosowań. (Dla tych czytelników, którzy nie czytają rozdziałów kolejno jeden po drugim: pierwsze argumenty ze słowami kluczowymi zostały opisane w rozdziale 18. Dla czytelników przerabiających załączone do książki przykłady: poniżej listingów znajdują się polecenia zawierające nazwy plików, w których te listingi się znajdują).

```
class tracer:                      # Stan w atrybutach instancji

    def __init__(self, func):      # W momencie dekoracji @

        self.calls = 0              # Zapisanie func dla późniejszego wywołania

        self.func = func

    def __call__(self, *args, **kwargs):  # W momencie wywołania oryginalnej
funkcji

        self.calls += 1

        print('wywołanie %s %s' % (self.calls, self.func.__name__))

        return self.func(*args, **kwargs)

@tracer

def spam(a, b, c):                  # To samo co: spam = tracer(spam)

    print(a + b + c)                # Uruchamia tracer.__init__


@tracer
```

```

def eggs(x, y):                      # To samo co: eggs = tracer(eggs)
    print(x ** y)                    # Opakowuje eggs w obiekt tracer
spam(1, 2, 3)                        # Tak naprawdę wywołuje instancję tracer:
wykonuje tracer.__call__             # spam jest atrybutem instancji
spam(a=4, b=5, c=6)                  # Tak naprawdę wywołuje instancję tracer,
eggs(2, 16)                          # self.func to eggs
eggs(4, y=4)                         # self.calls zliczane tutaj per funkcja
(potrzebna instrukcja nonlocal z 3.0)

```

Tak jak wersja oryginalna, powyższy kod wykorzystuje *atrybuty instancji klasy* do zapisania stanu w sposób jawnym. Zarówno opakowana funkcja, jak i licznik wywołań są informacjami *per instancję* — każda dekoracja otrzymuje ich własną kopię. Po wykonaniu skryptu w Pythonie 2.x lub 3.x wynik powyższej wersji kodu będzie następujący. Warto zwrócić uwagę na to, że funkcje `spam` oraz `eggs` mają własne liczniki wywołań, ponieważ każda dekoracja tworzy nową instancję klasy.

wywołanie 1 spam

6

wywołanie 2 spam

15

wywołanie 1 eggs

65536

wywołanie 2 eggs

256

Choć przydaje się przy dekoracji funkcji, taki schemat kodu sprawia pewne problemy po zastosowaniu do metod (więcej informacji na ten temat nieco później).

Zakresy zawierające oraz zmienne globalne

Ten sam efekt mogą dawać również referencje do zakresów zawierających instrukcji `def` oraz zagnieżdżone instrukcje `def`, zwłaszcza w przypadku danych statycznych, takich jak udekorowana oryginalna funkcja. W poniższym przykładzie potrzebny był nam jednak również licznik w zakresie zawierającym, który *zmienia się* z każdym wywołaniem, co nie jest możliwe w Pythonie 2.x (jak pamiętamy z rozdziału 17., instrukcja `nonlocal` jest dostępna tylko w wersji 3.x).

W wersji 2.x możemy wciąż używać klas i atrybutów w sposób opisany w poprzednim podrozdziale, ale są też inne opcje. Jedną z nich, dostępną w wersjach 2.x i 3.x, jest przeniesienie zmiennych stanu do *globalnego zakresu* z deklaracjami:

```

calls = 0

def tracer(func):                   # Stan w zakresie zawierającym oraz
zmiennej globalnej

    def wrapper(*args, **kwargs):   # Zamiast atrybutów klas
        global calls               # Zmienna calls jest globalna, a nie per
funkcja

        calls += 1

```

```

        print('wywołanie %s %s' % (calls, func.__name__))
    return func(*args, **kwargs)

    return wrapper

@tracer
def spam(a, b, c):                      # To samo co: spam = tracer(spam)
    print(a + b + c)

@tracer
def eggs(x, y):                         # To samo co: eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3)                           # Tak naprawdę wywołuje wrapper,
dowiązany do func

spam(a=4, b=5, c=6)                     # wrapper wywołuje spam

eggs(2, 16)                            # Tak naprawdę wywołuje wrapper,
dowiązany do eggs

eggs(4, y=4)                            # Zmienna globalna calls nie jest tutaj
liczona per funkcja!

```

Niestety, przeniesienie licznika do wspólnego zakresu globalnego, tak by mógł on być modyfikowany w ten sposób, oznacza również, że będzie on *współdzielony* przez każdą opakowaną funkcję. W przeciwnieństwie do atrybutów instancji klas liczniki globalne są wspólne dla programu, a nie per funkcja — licznik inkrementowany jest z każdym wywołaniem śledzonej funkcji. Różnica staje się zauważalna, jeśli porównamy wynik tej wersji z wynikiem poprzedniej — jeden współdzielony, globalny licznik wywołań jest niepoprawnie uaktualniany przez wywołania każdej udekorowanej funkcji.

```

c:\code> python decorator3.py
wywołanie 1 spam
6
wywołanie 2 spam
15
wywołanie 3 eggs
65536
wywołanie 4 eggs
256

```

Zakresy funkcji zawierających oraz zmienne nielokalne

Współdzielony stan globalny może w pewnych przypadkach być tym, czego chcemy. Jeśli jednak potrzebny jest nam licznik *per funkcja*, możemy albo użyć klas (jak wcześniej), albo skorzystać z nowej instrukcji Pythona 3.x o nazwie `nonlocal`, opisanej w rozdziale 17. Ponieważ ta nowa instrukcja pozwala na modyfikowanie zmiennych z zakresu funkcji zawierającej, może służyć jako zmienne dane per dekoracja. Poniższy kod działa tylko w wersji 3.x:

```

def tracer(func):                      # Stan w zakresie zawierającym i
zmiennej nielokalnej

```

```

    calls = 0                                # Zamiast atrybutów klasy lub
zmiennej globalnej

    def wrapper(*args, **kwargs):            # Zmienna calls jest per funkcja, a
nie globalna

        nonlocal calls

        calls += 1

        print('wywołanie %s %s' % (calls, func.__name__))

        return func(*args, **kwargs)

    return wrapper

@tracer

def spam(a, b, c):                         # To samo co: spam = tracer(spam)

    print(a + b + c)

@tracer

def eggs(x, y):                            # To samo co: eggs = tracer(eggs)

    print(x ** y)

spam(1, 2, 3)                             # Tak naprawdę wywołuje wrapper,
dowiązany do func

spam(a=4, b=5, c=6)                        # wrapper wywołuje spam

eggs(2, 16)                               # Tak naprawdę wywołuje wrapper,
dowiązany do eggs

eggs(4, y=4)                               # Zmienna nielokalna calls nie jest
tutaj per funkcja

```

Teraz, ponieważ zmienne z zakresu funkcji zawierającej nie są globalne dla całego programu, każda opakowana funkcja otrzymuje znowu własny licznik, tak jak to było w przypadku klas i atrybutów. Oto wynik wykonania powyższego kodu w Pythonie 3.x:

```
c:\code> py -3 decorator4.py
wywołanie 1 spam
6
wywołanie 2 spam
15
wywołanie 1 eggs
65536
wywołanie 2 eggs
256
```

Atrybuty funkcji

Wreszcie, jeśli nie używamy Pythona 3.x i nie mamy dostępu do instrukcji `nonlocal` albo chcemy, aby kod działał *zarówno* w wersji 2.x, jak i 3.x, nadal możemy ominąć zmienne globalne oraz klasy, na potrzeby zmiennego stanu wykorzystując *atrybuty funkcji*. W nowszych wersjach

Pythona, począwszy od 2.1, możemy za pomocą zapisu `funkcja.atrybut=wartość` przypisywać dowolne atrybuty do funkcji w celu ich dołączania. Ponieważ funkcja fabryczna przy każdym wywołaniu tworzy nową funkcję, więc jej atrybuty przechowują stan. Co więcej, ta technika jest potrzebna tylko w przypadku stanów, które muszą się zmieniać. Odwołania otaczające zakres są zachowywane i działają automatycznie.

W naszym przykładzie możemy po prostu na potrzeby zapisania stanu wykorzystać kod `wrapper.calls`. Poniższy kod działa tak samo jak poprzednia wersja ze zmiennymi nielokalnymi, ponieważ licznik znów działa per udekorowana funkcja. Jedyną różnicą jest to, że rozwiążanie to działa również w Pythonie 2.x.

```
def tracer(func):                      # Stan w zakresie funkcji zawierającej i
    atrybucie funkcji

    def wrapper(*args, **kwargs):      # Zmienna calls jest per funkcja, a nie
        globalna

        wrapper.calls += 1

        print('wywołanie %s %s' % (wrapper.calls, func.__name__))

        return func(*args, **kwargs)

    wrapper.calls = 0

    return wrapper

@tracer

def spam(a, b, c):                    # To samo co: spam = tracer(spam)
    print(a + b + c)

@tracer

def eggs(x, y):                      # To samo co: eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3)                        # Tak naprawdę wywołuje wrapper,
dowiązany do spam

spam(a=4, b=5, c=6)                  # wrapper wywołuje spam

eggs(2, 16)                          # Tak naprawdę wywołuje wrapper,
dowiązany do eggs

eggs(4, y=4)                         # Zmienna wrapper.calls działa tutaj per
funkcja
```

Jak wiemy z rozdziału 17., rozwiążanie to działa tylko dzięki temu, że nazwa `wrapper` zachowywana jest w zakresie funkcji zawierającej `tracer`. Gdy później inkrementujemy zmienną `wrapper.calls`, nie zmieniamy samej zmiennej `wrapper`, dlatego deklaracja `nonlocal` nie jest wymagana. Powyższy kod działa poprawnie w każdej wersji Pythona.

```
c:\code> py -2 decorator5.py
```

```
...wynik taki sam jak poprzednio, ale kod działa również w wersji 2.x...
```

Powyższe rozwiązanie niemalże trafiło do przypisu, ponieważ jest mniej oczywiste niż deklaracja `nonlocal` z Pythona 3.x i lepiej jest zostawić je na potrzeby przypadków, w których inne rozwiązania zawodzą. Jednak atrybuty funkcji mają niezaprzeczalne zalety. Po pierwsze dają dostęp do zapisanego stanu *spoza* kodu dekoratora. Zmienne nielokalne są widoczne tylko wewnątrz zagnieżdżonej funkcji, natomiast atrybuty funkcji mają szerszy zakres widoczności.

Po drugie adresy funkcji są bardziej *uniwersalne*. Opisany schemat działa również w wersji 2.x, jest zatem niezależny od wersji.

Atrybuty funkcji wykorzystamy jednak w odpowiedzi do jednego z pytań kończących rozdział, w którym ich widzialność poza obiektem wywoływalnym jest zaletą. Jako zmienne stanu skojarzone z kontekstem, w którym są stosowane, są odpowiednikami zmiennych nielokalnych w otaczającym zakresie. Jak zwykle, wybór jednego z wielu dostępnych narzędzi jest nieodłączną częścią programowania.

Ponieważ dekoratory często wymuszają kilka poziomów obiektów wywoływalnych, możemy połączyć funkcje z zakresami zawierającymi i klasami z atrybutami w celu uzyskania różnych rodzajów struktur kodu. Jak jednak zobaczymy nieco później, czasami różnice mogą być bardziej subtelne, niż moglibyśmy się tego spodziewać — każda udekorowana funkcja powinna mieć własny stan, a każda udekorowana klasa może wymagać stanu zarówno dla siebie samej, jak i dla każdej wygenerowanej instancji.

Tak naprawdę, jak zobaczymy w kolejnym podrozdziale, jeśli chcemy zastosować dekoratory funkcji także do metod klas, musimy również uważać na rozróżnienie, które Python robi pomiędzy dekoratorami zapisanymi w postaci wywoływalnych obiektów instancji klas a dekoratorami zapisanymi w postaci funkcji.

Uwagi na temat klas I — dekorowanie metod klas

Kiedy napisałem pierwszy dekorator funkcji `tracer` zamieszczony powyżej w pliku `decorator1.py`, naiwnie założyłem, że można go również zastosować do dowolnej *metody* — udekorowane metody powinny działać tak samo, jednak automatyczny argument instancji `self` miałby zostać dołączony na początku `*args`. Niestety, *nie miałem racji* — po zastosowaniu do metody klasy pierwsza wersja dekoratora `tracer` nie działa, ponieważ `self` jest instancją klasy dekoratora, a instancja udekorowanej klasy podmiotowej nie zostaje dołączona do `*args`. Takie zachowanie występuje zarówno w Pythonie 3.x, jak i 2.x.

Zjawisko to omówiłem pokrótce we wcześniejszej części rozdziału. Teraz jednak zobaczymy je w kontekście realistycznego, działającego kodu. Gdy mamy dekorator śledzący oparty na klasie:

```
class tracer:

    def __init__(self, func):                      # W momencie dekoracji @
        self.calls = 0                               # Zapisanie func dla
                                                       # późniejszego wywołania

        self.func = func

    def __call__(self, *args, **kwargs):           # W momencie wywołania
        oryginalnej funkcji

        self.calls += 1

        print('wywołanie %s %s' % (self.calls, self.func.__name__))

        return self.func(*args, **kwargs)
```

dekoracja prostych funkcji działa zgodnie z wcześniejszym opisem:

```
@tracer

def spam(a, b, c):                            # spam = tracer(spam)
                                                # Uruchamia tracer.__init__
    print(a + b + c)

>>> spam(1, 2, 3)
```

```
Wywołanie 1 spam
6
>>> spam(a=4, b=5, c=6)                      # spam zapisany w atrybucie
instancji
```

Wywołanie 2 spam

15

Dekoracja metody klasy nadal jednak nie działa (bardziej czujni Czytelnicy rozpoznają w tym przykładzie klasę Person wziętą z omówienia programowania zorientowanego obiektowo w rozdziale 28.).

```
class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    @tracer
    def giveRaise(self, percent):          # giveRaise = tracer(giveRaise)
        self.pay *= (1.0 + percent)
    @tracer
    def lastName(self):                  # lastName = tracer(lastName)
        return self.name.split()[-1]
>>> bob = Person('Robert Zielony', 50000)      # tracer pamięta funkcje metod
>>> bob.giveRaise(.25)                      # Wykonuje tracer.__call__(???, .25)

wywołanie 1 giveRaise
TypeError: giveRaise() missing 1 required positional argument: 'percent'
>>> print(bob.lastName())                   # Wykonuje tracer.__call__(???)
```

wywołanie 1 lastName

TypeError: lastName() missing 1 required positional argument: 'self'

Sedno problemu leży w argumencie `self` metody `__call__` klasy `tracer` — czy jest to instancja klasy `tracer`, czy może instancja klasy `Person`? W obecnej postaci kodu potrzebujemy *obu* — instancji `tracer` dla stanu dekoratora, natomiast instancji `Person` dla przekierowania do oryginalnej metody. Tak naprawdę `self` *musi* jednak być obiektem `tracer`, tak byśmy mogli uzyskać dostęp do informacji o stanie klasy `tracer`. Będzie to prawdziwe bez względu na to, czy dekorujemy prostą funkcję, czy też metodę.

Niestety, gdy nazwa naszej udekorowanej metody zostaje ponownie dowiązana do obiektu instancji klasy za pomocą metody `__call__`, Python przekazuje do `self` jedynie *instancję* `tracer`. Nie przekazuje w liście argumentów elementu `Person`. Co więcej, ponieważ klasa `tracer` nie wie nic o instancji `Person`, którą próbujemy przetwarzać za pomocą wywołań metod, nie ma możliwości utworzenia metody dowiązanej do instancji i tym samym nie da się w sposób poprawny wykonać wywołania. Nie jest to błąd, tylko niezwykle subtelna zawiłość.

Tak naprawdę powyższa wersja kodu przekazuje zbyt małą liczbę argumentów do udekorowanej metody i kończy się zwróceniem błędu. Można to zweryfikować, dodając do metody `__call__` dekoratora wiersz wyświetlający wszystkie argumenty. Jak widać, `self` to instancja klasy `tracer`, natomiast instancji klasy `Person` w ogóle nie ma.

```
>>> bob.giveRaise(.25)
<__main__.tracer object at 0x02D6AD90> (0.25,) {}
wywołanie 1 giveRaise
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 9, in __call__
      TypeError: giveRaise() missing 1 required positional argument: 'percent'
```

Jak wspomniano wcześniej, ma to miejsce, gdyż Python przekazuje domniemaną instancję podmiotową do `self`, gdy nazwa metody dowiązywana jest jedynie do prostej funkcji. Kiedy jest to instancja klasy wywoływalnej, zamiast tego przekazywana jest instancja tej klasy. Python tworzy obiekt dowiązanej metody zawierający podmiotową instancję, kiedy metoda jest prostą funkcją, a nie wywoływalną instancją innej klasy.

Wykorzystywanie zagnieżdzonych funkcji do dekoracji metod

Jeśli chcemy, by dekoratory funkcji działały zarówno na prostych funkcjach, jak i na metodach klas, najprostszym rozwiązaniem jest wykorzystanie jednego z opisanych wcześniej rozwiązań dla zachowywania stanu. Dekorator funkcji należy zapisać w kodzie w postaci zagnieżdzonych instrukcji `def`, tak byśmy nie musieli polegać na pojedynczym argumencie instancji `self`, który ma być zarówno instancją klasy opakowującej, jak i instancją klasy podmiotowej.

Poniższe rozwiązanie alternatywne naprawia nasz błąd, wykorzystując do tego zmienne nielokalne z Pythona 3.x. Aby działał w wersji 2.x, należy go zmienić, tj. wykorzystać atrybuty zmienianej funkcji. Ponieważ udekorowane metody są ponownie dowiązywane do prostych funkcji zamiast do obiektów instancji, Python w poprawny sposób przekazuje obiekt klasy `Person` jako pierwszy argument, a dekorator przekazuje go w pierwszym argumencie listy `*args` do argumentu `self` prawdziwych, udekorowanych metod.

```
# Dekorator przeznaczony zarówno dla funkcji, jak i dla metod
def tracer(func):                      # Użycie funkcji, a nie klasy z __call__
    calls = 0                            # Inaczej "self" będzie jedynie instancją
    dekoratora!
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('wywołanie %s %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall
if __name__ == '__main__':
    # Ma zastosowanie do prostych funkcji
    @tracer
```

```

def spam(a, b, c):          # spam = tracer(spam)
    print(a + b + c)         # onCall pamięta spam

@tracer

def eggs(N):
    return 2 ** N

spam(1, 2, 3)               # Wykonuje onCall(1, 2, 3)
spam(a=4, b=5, c=6)
print(eggs(32))

# Ma zastosowanie również do funkcji metod klas!

class Person:

    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer

    def giveRaise(self, percent):      # giveRaise = tracer(giveRaise)
        self.pay *= (1.0 + percent)   # onCall pamięta giveRaise

    @tracer

    def lastName(self):             # lastName = tracer(lastName)
        return self.name.split()[-1]

print('metody...')

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)

print(bob.name, anna.name)
anna.giveRaise(.10)              # Wykonuje onCall(anna, .10)

print(anna.pay)
print(bob.lastName(), anna.lastName())  # Wykonuje onCall(bob), lastName w
                                         zakresach

```

Samotestujący kod został wcięty i umieszczony pod instrukcją sprawdzającą atrybut `__name__`, dzięki czemu dekorator można importować i wykorzystywać w innych kodach. Powyższa wersja działa w ten sam sposób zarówno na funkcjach, jak i na metodach, ale tylko w wersji 3.x, ponieważ wykorzystuje instrukcję `nonlocal`:

```
c:\code> py -3 calltracer.py
```

```
wywołanie 1 spam
```

```
6
```

```
wywołanie 2 spam
```

```
15
```

```
wywołanie 1 eggs
4294967296
metody...
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona
```

Aby mieć pewność, że model jest zrozumiały, warto prześledzić powyższe wyniki. W kolejnym podrozdziale opisane jest alternatywne rozwiązywanie dotyczące klas. Jest jednak znacznie bardziej skomplikowane.

Wykorzystywanie deskryptorów do dekorowania metod

Choć rozwiązanie z zagnieżdżonymi funkcjami przedstawione wyżej jest najprostszym sposobem obsługiwanego dekoratorów mających zastosowanie zarówno do funkcji, jak i do metod klas, możliwe są również inne rozwiązania. Mogą nam tutaj pomóc na przykład *deskryptory* omówione w poprzednim rozdziale.

Przypomnijmy, że zgodnie z informacjami z poprzedniego rozdziału deskryptor może być atrybutem klasy przypisywanym do obiektów za pomocą metody `__get__` wykonywanej automatycznie wtedy, gdy następuje odwołanie się do atrybutu oraz jego pobranie (w Pythonie 2.x niezbędne jest dziedziczenie klasy `object`, które nie jest wymagane w wersji 3.x).

```
class Descriptor(object):
    def __get__(self, instance, owner): ...
class Subject:
    attr = Descriptor()
X = Subject()
X.attr                      # Wykonuje w przybliżeniu
Descriptor.__get__(Subject.attr, X, Subject)
```

Deskryptory mogą również zawierać metody dostępu `__set__` oraz `__del__`, których tutaj jednak nie potrzebujemy. Ponieważ metoda `__get__` deskryptora otrzymuje po wywołaniu instancję klasy deskryptora, jak i klasy podmiotowej, dobrze nadaje się do dekorowania metod, kiedy przy wywołaniach potrzebne są nam zarówno stan dekoratora, jak i instancja oryginalnej klasy. Rozważmy następujący wariant dekoratora śledzącego, będącego również deskryptorem, jeżeli zastosuje się go na poziomie metod klasy:

```
class tracer(object):                      # Dekorator i deskryptor
    def __init__(self, func):                # W momencie dekoracji @
        self.calls = 0                       # Zapisanie func na potrzeby
        późniejszego wywołania
        self.func = func
    def __call__(self, *args, **kwargs):      # W momencie wywołania
        oryginalnej funkcji
```

```

        self.calls += 1
        print('wywołanie %s %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)

    def __get__(self, instance, owner):           # W momencie pobrania
        atrybutu metody
            return wrapper(self, instance)

    class wrapper:
        def __init__(self, desc, subj):          # Zapisanie obu instancji
            self.desc = desc                   # Przekierowanie wywołań z
            powrotem do dekoratora
            self.subj = subj

        def __call__(self, *args, **kwargs):
            return self.desc(self.subj, *args, **kwargs)   # Wykonuje
            tracer.__call__

        @tracer
        def spam(a, b, c):                      # spam = tracer(spam)
            ...to samo co poprzednio...          # Wykorzystuje jedynie
            __call__

        class Person:
            @tracer
            def giveRaise(self, percent):         # giveRaise =
            tracer(giveRaise)
                ...to samo co poprzednio...      # Sprawia, że giveRaise
                staje się deskryptorem

```

Takie rozwiązanie działa tak samo jak poprzedni kod funkcji zagnieżdzonej. Wykonywane operacje są różne w zależności od kontekstu:

- Udekorowane *funkcje* wywołują jedynie metodę `__call__`, natomiast nowsze wywołują `__get__`.
- Udekorowane *metody* wywołują najpierw metodę `__get__` w celu przeanalizowania pobrania nazwy metody (dla *instancja.metoda*). Obiekt zwracany przez metodę `__get__` zachowuje instancję klasy podmiotowej i jest następnie wywoływany w celu zakończenia wyrażenia wywołania, uruchamiając metodę `__call__` (dla *(args...)*).

Przykładowo poniższe wywołanie kodu testu:

```

anna.giveRaise(.10)                         # Wykonuje __get__, a następnie
__call__

```

wykonuje najpierw metodę `tracer.__get__`, ponieważ atrybut `giveRaise` klasy `Person` został ponownie dowiązany do deskryptora za pomocą dekoratora funkcji. Wyrażenie wywołania uruchamia następnie metodę `__call__` zwracanego obiektu opakowującego, która z kolei wywołuje metodę `tracer.__call__`. Innymi słowy, wywołanie udekorowanej metody uruchamia czteroetapowy proces: wywołanie `tracer.__get__`, po nim wywołanie metod `wrapper.__call__` i `tracer.__call__`, a na koniec wywołanie oryginalnej, opakowanej metody.

Obiekt `wrapper` zachowuje zarównoinstancję deskryptora, jak i instancję podmiotową, dlatego może przekazać sterowanie z powrotem do instancji oryginalnej klasy dekoratora (czy deskryptora). W rezultacie obiekt `wrapper` zapisuje instancję podmiotowej klasy dostępną w trakcie pobrania atrybutu metody i dodaje ją do listy argumentów późniejszego wywołania, przekazywanej do metody `__call__`. Przekierowanie wywołania w ten sposób z powrotem do instancji klasy deskryptora wymagane jest w tej aplikacji, by wszystkie wywołania opakowanej metody wykorzystywały te same informacje o stanie licznika `calls` w obiekcie instancji deskryptora.

Alternatywnie moglibyśmy wykorzystać funkcję zagnieżdzoną i referencje do zakresu funkcji zawierającej w celu uzyskania tego samego efektu. Poniższa wersja działa tak samo jak poprzednia, zamieniając jednak klasę i atrybuty obiektów na funkcję zagnieżdzoną i referencje do zakresu. Wymaga też wyraźnie mniejszej ilości kodu.

```
class tracer(object):

    def __init__(self, func):
        self.calls = 0
        self.func = func

    def __call__(self, *args, **kwargs):
        self.calls += 1
        print('wywołanie %s %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)

    def __get__(self, instance, owner):
        def wrapper(*args, **kwargs):
            return self(instance, *args, **kwargs)
        return wrapper

    def wrapper(*args, **kwargs):
        return self(*args, **kwargs)

    return wrapper
```

By prześledzić dwuetapowy proces pobierania i wywoływania, można samodzielnie dodać do metod alternatywnych instrukcję `print` i wykonać je z tym samym kodem testowym co w przypadku pokazanej wcześniej wersji z funkcją zagnieżdzoną (zawartej w pliku `calltracer-descr.py`). W obu przypadkach rozwiązanie oparte na deskryptorze jest bardziej subtelne od opcji z funkcją zagnieżdzoną, dlatego też będzie raczej drugim wyborem. Trzeba jednak przyznać, że jeżeli złożoność tego rozwiązania nie przeraża, to koszt wydajności na pewno! Ten wzorzec kodu może się jednak przydać w innych kontekstach.

Warto również wspomnieć, że taki dekorator oparty na deskryptorze można zakodować w prostszy sposób, który jednak dotyczy tylko metod, a nie prostych funkcji. Wynika to z wewnętrznego ograniczenia deskryptorów atrybutów (i stanowi negację problemu, który chcieliśmy rozwiązać: znaleźć dekorator zarówno dla funkcji, jak i metod).

```
class tracer(object):
    def __init__(self, meth):
        self.calls = 0
        self.meth = meth

    def __get__(self, instance, owner):
        def wrapper(*args, **kwargs):
            self.calls += 1
            print('wywołanie %s %s' % (self.calls, self.meth.__name__))
            return self.meth(instance, *args, **kwargs)
        return wrapper
```

```

def wrapper(*args, **kwargs):                      # W momencie wywołania metody
    self.calls += 1
    print('wywołanie %s %s' % (self.calls, self.meth.__name__))
    return self.meth(instance, *args, **kwargs)

return wrapper

class Person:
    @tracer                                         # Dotyczy metod klasy
    def giveRaise(self, percent):                   # giveRaise =
tracer(giveRaise)
    ...
deskryptor                                         # Przekształcenie giveRaise w
                                                       # Nie działa w przypadku
@tracer                                         # Nie działa w przypadku
prostych funkcji
    def spam(a, b, c):                           # spam = tracer(spam)
    ...
żaden atrybut                                     # Tutaj nie jest pobierany

```

W dalszej części rozdziału będziemy raczej wykorzystywać klasy lub funkcje w celu zapisania w kodzie dekoratorów funkcji, o ile będą one miały zastosowanie jedynie do funkcji. Niektóre dekoratory mogą nie wymagać instancji oryginalnej klasy i nadal będą działały zarówno na funkcjach, jak i metodach, jeśli zapiszemy je w postaci klas. Coś takiego jak własny dekorator `staticmethod` Pythona nie wymaga na przykład instancji klasy podmiotowej (i tak naprawdę jedynym jego celem jest usunięcie instancji z wywołania).

Morał płynący z tej historii jest jednak taki, że jeśli chcemy, by nasze dekoratory działały zarówno na prostych funkcjach, jak i na metodach klas, lepiej będzie wykorzystać przedstawiony tutaj wzorzec kodu oparty na funkcji zagnieżdżonej zamiast klasy z przechwytywaniem wywołań.

Mierzenie czasu wywołania

By w nieco szerszym zakresie wypróbować możliwości dekoratorów funkcji, przejdźmy do innego przypadku użycia. Nasz kolejny dekorator mierzy czas trwania wywołań udekorowanej funkcji — zarówno dla pojedynczego wywołania, jak i całkowity czas dla wszystkich wywołań. Dekorator zostanie zastosowany do dwóch funkcji w celu porównania czasu wymaganego do tworzenia listy składanej oraz wywołania funkcji wbudowanej `map`.

```

# Plik timerdecor.py

# Uwaga: instrukcja range zwraca listę w wersji 2.x, a iterowalny obiekt w
wersji 3.x

# Uwaga: tak zakodowana klasa timer nie działa w przypadku metod (patrz
rozwiązanie quizu)

import time, sys

force = list if sys.version_info[0] == 3 else (lambda X: X)

class timer:
    def __init__(self, func):

```

```

        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kargs):
        start = time.clock()
        result = self.func(*args, **kargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('%s: %.5f, %.5f' % (self.func.__name__, elapsed, self.alltime))
        return result
@timer
def listcomp(N):
    return [x * 2 for x in range(N)]
@timer
def mapcall(N):
    return force(map((lambda x: x * 2), range(N)))
result = listcomp(5)                                # Czas dla tego wywołania, wszystkich
wywołań, zwracana wartość
listcomp(50000)
listcomp(500000)
listcomp(1000000)
print(result)
print('allTime = %s' % listcomp.alltime)      # Całkowity czas dla wszystkich
wywołań listcomp
print('')
result = mapcall(5)
mapcall(50000)
mapcall(500000)
mapcall(1000000)
print(result)
print('allTime = %s' % mapcall.alltime)      # Całkowity czas dla wszystkich
wywołań mapcall
print('**map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))

```

Po uruchomieniu powyższego kodu w wersji Pythona 2.x lub 3.x otrzymamy przedstawiony niżej wynik zawierający nazwę wywoływanej funkcji, czas jej wykonania, całkowity czas dotychczasowych wywołań, wynik zwrocony po pierwszym wywołaniu, całkowity czas wszystkich wywołań i na końcu stosunek czasów wywołań obu funkcji.

c:\code> py -3 timerdeco1.py

```
listcomp: 0.00001, 0.00001
listcomp: 0.00499, 0.00499
listcomp: 0.05716, 0.06215
listcomp: 0.11565, 0.17781
[0, 2, 4, 6, 8]
allTime = 0.17780527629411225
mapcall: 0.00002, 0.00002
mapcall: 0.00988, 0.00990
mapcall: 0.10601, 0.11591
mapcall: 0.21690, 0.33281
[0, 2, 4, 6, 8]
allTime = 0.3328064956447921
**map/comp = 1.872
```

Uzyskane wyniki zależą od wersji Pythona i szybkości komputera. Sumaryczny czas jest tutaj atrybutem instancji klasy. Jak zwykle mapa jest niemal dwukrotnie wolniejsza od listy, która nie wywołuje żadnych funkcji (oznacza to, że mapa jest wolniejsza, ponieważ musi wywoływać funkcję).

Dekoratory a pomiar czasu wywołania

Dla porównania w rozdziale 21. został opisany sposób pomiaru czasu wykonania iteracji *bez użycia dekoratorów*. Dla przypomnienia: użyte były tam dwie techniki umożliwiające mierzenie czasu pojedynczych wywołań — jedna własnego pomysłu, a druga wykorzystująca specjalną bibliotekę. Poniżej techniki te zostały użyte do pomiaru czasu przetwarzania wyrażenia listowego iterującego milion razy kod testujący dekorator. Dodatkowy koszt wprowadził kod zarządzający zawierający zewnętrzną pętlę i wywołania funkcji:

```
>>> def listcomp(N): [x * 2 for x in range(N)]
>>> import timer # Chapter 21 techniques
>>> timer.total(1, listcomp, 1000000)
(0.1461295268088542, None)
>>> import timeit
>>> timeit.timeit(number=1, stmt=lambda: listcomp(1000000))
0.14964829430189397
```

W tym przypadku rozwiązanie bez dekoratora pozwalałoby na wykorzystanie funkcji podmiotowych z pomiarem czasu bądź bez niego, jednak skomplikowałoby także sygnaturę wywołania, gdyby pomiar czasu był pożądany (musielibyśmy dodawać kod z każdym wywołaniem zamiast raz w instrukcji `def`). Co więcej, sposób bez użycia dekoratora nie dawałby bezpośrednio gwarancji, że wszystkie wywołania budujące listę w programie zostały przekierowane przez logikę pomiaru czasu — poza próbą odnalezienia wszystkich i potencjalnego zmodyfikowania ich. Z tego powodu trudno byłoby zmierzyć skumulowany czas wszystkich wywołań.

Zazwyczaj *dekoratory* warto stosować wtedy, gdy funkcje są już wdrożone, stanowią części większego systemu i trudno byłoby je analizować za pomocą innych funkcji w takcie działania.

Z drugiej strony, ponieważ dekoratory modyfikują każdą funkcję, wprowadzając do niej kody pomiarowe, podejście bez dekoratorów może okazać się lepsze, gdy trzeba monitorować wybrane wywołania. Jak zawsze, różne narzędzia pełnią różne role.



Kompatybilność pomiaru czasu i nowe opcje w wersji 3.3: w rozdziale 21. opisane jest dokładniej wybieranie i stosowanie funkcji z modułu `time`, w tym pobięźnie nowe i ulepszone funkcje, które pojawiły się w wersji 3.3 (np. `perf_counter`). Tutaj, aby opis był zwięzły, a kod niezależny od wersji, zastosowaliśmy uproszczone podejście. Jednak funkcja `perf_counter` może nie najlepiej się sprawdzać, nawet w wersjach starszych niż 3.3. Dodatkowo może okazać się konieczne wykonanie testów w innym systemie niż Windows i w innej wersji Pythona.

Nieuwanse pomiaru czasu

Należy zwrócić uwagę, że w skrypcie użyta została instrukcja `force`, dzięki której kod może działać w wersjach 2.x oraz 3.x. Jak pisałem w rozdziale 14., wbudowana funkcja `map` zwraca obiekt iterowalny, który w wersji 3.x zwraca wyniki na żądanie, a w wersji 2.x zwraca je w postaci listy. Dlatego nie można wprost porównywać mapy z listą. W rzeczywistości, jeżeli funkcji `map` nie zapakuje się do listy w celu uzyskania wyników, jest ona w wersji 3.x wykonywana niemal natychmiast. Funkcja zwraca obiekt iterowalny bez iterowania!

Natomiast użycie listy w wersji 2.x dodatkowo niesprawiedliwie obciąża funkcję `map`. Wyniki zawierałyby w takim przypadku czasy utworzenia dwóch list, a nie jednej. Aby tego uniknąć, skrypt wybiera odpowiednią funkcję opakowującą listę w zależności od wersji Pythona, odczytaną za pomocą modułu `sys`. W wersji 3.x jest to funkcja `list`, a w wersji 2.x funkcja, która nie wykonuje żadnych operacji i po prostu zwraca niezmienioną wartość argumentu. Wprowadza to wprawdzie niewielką stałą zwłokę, ale pomijalnie małą w porównaniu z czasem wykonania iteracji w mierzonej funkcji.

Choć opisane podejście powoduje, że porównanie czasu wykonania wyrażenia listowego i funkcji `map` w wersjach 2.x i 3.x jest bardziej rzetelne, ponieważ w wersji 3.x `range` jest również iteratorem, nie można wprost porównywać ze sobą wyników uzyskiwanych w obu wersjach, chyba że wywołanie wyniesie się poza mierzony kod. Można jednak dokonywać względnych porównań, a wyniki będą odzwierciedlać dobre praktyki stosowane w obu wersjach Pythona. Jednak iteracja `range` wprowadza dodatkowe opóźnienie tylko w wersji 3.x. Więcej informacji na ten temat zawartych jest w rozdziale 21. poświęconym tworzeniu wzorców odniesienia. Generowanie porównywalnych wyników jest często nietrywialnym zadaniem.

Wreszcie podobnie jak w przypadku dekoratorów śledzących: aby móc korzystać z dekoratora mierzącego czas w innych modułach, powinniśmy dokonać indentacji kodu testu samosprawdzającego na końcu pliku pod testem `__name__`, tak by wykonywany był on tylko przy wykonywaniu pliku, a nie jego importowaniu. Nie będziemy jednak tego robić, ponieważ za moment dodamy do naszego kodu dodatkowe opcje.

Dodawanie argumentów dekoratora

Dekorator mierzący czas z poprzedniego podrozdziału działa, ale byłoby miło, gdyby był bardziej konfigurowalny. Podanie etykiety danych wyjściowych czy na przykład włączanie i wyłączanie komunikatów śledzenia mogłyby się przydać w tego typu uniwersalnym narzędziu. W takiej sytuacji przydają się argumenty dekoratora — po poprawnym dodaniu ich do kodu możemy je wykorzystać w celu określenia opcji konfiguracyjnych, które mogą się różnić dla każdej udekorowanej funkcji. Etyketę można na przykład dodać w następujący sposób:

```
def timer(label=''):  
    def decorator(func):
```

```

def onCall(*args):                                # Argumenty przekazane do
funkcji

    ...
# Funkcja zachowana w zakresie
zawierającym

    print(label, ...)                            # Etykieta zachowana w zakresie
zawierającym

    return onCall

    return decorator                            # Zwraca sam dekorator

@timer('==>')
(listcomp)

def listcomp(N): ...                            # Nazwa listcomp ponownie
dowiązana do dekoratora

listcomp(...)                                    # Tak naprawdę wywołuje
dekorator

```

Powyższy kod dodaje zakres funkcji zawierającej w celu zachowania argumentu dekoratora do zastosowania w późniejszym wywołaniu. Kiedy definiowana jest funkcja `listcomp`, tak naprawdę wywołuje ona funkcję `decorator` (wynik funkcji `timer`, wykonanej zanim nastąpi sama dekoracja) z wartością etykiety `label` dostępną w zakresie funkcji zawierającej. Oznacza to, że funkcja `timer` zwraca dekorator pamiętający zarówno argument dekoratora, jak i oryginalną funkcję oraz zwracający obiekt wywoływalny uruchamiający oryginalną funkcję w przypadku późniejszych wywołań. Ponieważ w ten sposób tworzony jest nowy dekorator i funkcja `onCall`, stan jest zachowywany tylko w obejmowanym zakresie.

Taką strukturę możemy wykorzystać w naszej funkcji mierzącej czas, tak by możliwe było przekazywanie etykiety oraz opcji sterowania śledzeniem w czasie dekoracji. Poniżej znajduje się przykład wykonujący to działanie, zapisany w module `timerdeco2.py`, tak by mógł on być importowany jako uniwersalne narzędzie. Do zachowania stanu wykorzystywana jest klasa, a nie zagnieżdzona funkcja, jednak ostateczny efekt jest taki sam:

```

import time

def timer(label='', trace=True):                  # Dla argumentów dekoratora:
zachowanie argumentów

    class Timer:
        def __init__(self, func):                # Dla @: zachowanie udekrowanej
funkcji
            self.func = func
            self.alltime = 0

        def __call__(self, *args, **kargs):      # Dla wywołań: wywołanie
oryginalnej funkcji
            start = time.clock()
            result = self.func(*args, **kargs)
            elapsed = time.clock() - start
            self.alltime += elapsed

            if trace:

```

```

        format = '%s %s: %.5f, %.5f'
        values = (label, self.func.__name__, elapsed, self.alltime)
        print(format % values)

    return result

return Timer

```

Większość naszej pracy polegała tutaj na osadzeniu oryginalnej klasy `Timer` w funkcji zawierającej w celu utworzenia zakresu zachowującego argumenty dekoratora. Zewnętrzna funkcja `timer` wywoływana jest przed wystąpieniem dekoracji i po prostu zwraca klasę `Timer`, która służy jako faktyczny dekorator. W momencie dekoracji tworzona jest instancja klasy `Timer`, pamiętająca samą udekorowaną funkcję i mająca również dostęp do argumentów dekoratora z zakresu funkcji zawierającej.

Pomiar czasu z użyciem argumentów dekoratora

Tym razem zamiast osadzać kod testu samosprawdzającego w pliku, wykonamy dekorator w innym pliku. Oto klient naszego dekoratora mierzącego czas — plik modułu `testseqs.py`, który ponownie zastosuje dekorator do alternatywnych rozwiązań w zakresie iteracji po sekwencjach.

```

import sys

from timerdeco2 import timer

force = list if sys.version_info[0] == 3 else (lambda X: X)

@timer(label='[CCC]==>')

def listcomp(N):
    # Jak listcomp = timer(...)

    (listcomp)

    return [x * 2 for x in range(N)]      # listcomp(...) uruchamia
    Timer.__call___.  

@timer(trace=True, label='[MMM]==>')

def mapcall(N):

    return map((lambda x: x * 2), range(N))

for func in (listcomp, mapcall):

    result = func(5)                      # Czas dla tego wywołania, wszystkich
    wywołań, zwracana wartość

        func(50000)
        func(500000)
        func(1000000)
        print(result)

        print('allTime = %s' % func.alltime)  # Całkowity czas dla wszystkich
    wywołań

    print('**map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))

```

I znów, jeśli chcemy w sprawiedliwy sposób wykonać test w Pythonie 3.x, musimy opakować funkcję `map` w wywołanie `list`. Po wykonaniu w obecnej postaci w Pythonie 3.x lub 2.x powyższy plik wyświetla następujące wyniki. Każda udekorowana funkcja ma teraz własną etykietę, zdefiniowaną za pomocą argumentów dekoratora. Jest to przydatne rozwiązanie w

przypadku, gdy w większych wynikach zwracanych przez program trzeba odszukać pomiary czasu.

```
c:\code> py -3 testseqs.py

[CCC]==> listcomp: 0.00001, 0.00001
[CCC]==> listcomp: 0.00504, 0.00505
[CCC]==> listcomp: 0.05839, 0.06344
[CCC]==> listcomp: 0.12001, 0.18344
[0, 2, 4, 6, 8]
allTime = 0.1834406801777564

[MMM]==> mapcall: 0.00003, 0.00003
[MMM]==> mapcall: 0.00961, 0.00964
[MMM]==> mapcall: 0.10929, 0.11892
[MMM]==> mapcall: 0.22143, 0.34035
[0, 2, 4, 6, 8]
allTime = 0.3403542519173618
**map/comp = 1.855
```

Jak zawsze, możemy także przetestować to w sesji interaktywnej w celu przekonania się, w jaki sposób działają argumenty konfiguracyjne.

```
>>> from mytools import timer
>>> @timer(trace=False)                                     # Brak śledzenia, zebranie
całkowitego czasu
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> listcomp.alltime
0.0037191417530599152
>>> listcomp
<timerdeco2.timer.<locals>.Timer object at 0x02957518>
>>> @timer(trace=True, label='\t=>')                      # Załączenie śledzenia
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
```

```
=> listcomp: 0.00106, 0.00106
>>> x = listcomp(5000)
=> listcomp: 0.00108, 0.00214
>>> x = listcomp(5000)
=> listcomp: 0.00107, 0.00321
>>> listcomp.alltime
0.003208920466562404
```

Powyższy dekorator funkcji mierzący czas można wykorzystać dla dowolnej funkcji, zarówno w modułach, jak i w sesji interaktywnej. Innymi słowy, automatycznie można go zakwalifikować jako *narzędzie ogólnego przeznaczenia* służące do pomiaru czasu wykonywania kodu w skryptach. Następny przykład argumentów dekoratora znajduje się w podrozdziale „Implementacja atrybutów prywatnych”, natomiast jeszcze kolejny — w podrozdziale „Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych”.



Metody pomiaru czasu: Dekorator timer z niniejszego podrozdziału działa na wszystkich *funkcjach*, jednak w celu zastosowania go również do *metod klas* wymagana jest niewielka modyfikacja. Mówiąc w skrócie, jak wspomnieliśmy we wcześniejszym podrozdziale zatytułowanym „Uwagi na temat klas I — dekorowanie metod klas”, należy unikać wykorzystywania zagnieżdzonej klasy. Ponieważ jednak taka zmiana będzie tematem pytań kończących rozdział, całkowicie pominę tutaj podanie pełnego rozwiązania.

Kod dekoratorów klas

Dotychczas pisaliśmy kod dekoratorów funkcji zarządzających wywołaniami funkcji, jednak, jak widzieliśmy, w Pythonie 2.x oraz 3.x dekoratory zostały rozszerzone w taki sposób, by działać również na klasach. Zgodnie z wcześniejszym opisem, choć są one podobne do dekoratorów funkcji, dekoratory klas stosowane są zamiast tego do klas — można je wykorzystać albo do zarządzania samymi *klasami*, albo do przechwytywania wywołań tworzących instancje w celu zarządzania *instancjami*. Podobnie do dekoratorów funkcji, dekoratory klas są tak naprawdę składnią opcjonalną, choć wiele osób uważa, że pozwalają one uczynić intencje programisty bardziej oczywistymi, a także minimalizują liczbę błędnych wywołań.

Klasy singletona

Ponieważ dekoratory klas mogą przechwytywać wywołania tworzące instancje, można je wykorzystać albo do zarządzania wszystkimi instancjami klasy, albo do rozszerzenia interfejsów tych instancji. By to zademonstrować, poniżej znajduje się przykład dekoratora klasy, który wykonuje to pierwsze zadanie — zarządzania instancjami klasy. Kod ten implementuje klasyczny wzorzec projektowy *singletona*, w którym istnieje maksymalnie jedna instancja klasy. Funkcja *singleton* definiuje i zwraca funkcję zarządzającą instancjami, natomiast składnia ze znakiem @ automatycznie opakowuje w tę funkcję klasę podmiotową.

```
# Globalna tabela w wersjach 3.x i 2.x
instances = {}

def singleton(aClass):
    # W momencie dekoracji @
```

```

def onCall(*args, **kwargs):
    # W momencie tworzenia instancji
    if aClass not in instances:
        # Jeden wpis słownika na klasę
        instances[aClass] = aClass(*args, **kwargs)
    return instances[aClass]
return onCall

By wykorzystać powyższy kod, należy udekorować klasy, dla których chcemy wymusić model z
pojedynczą instancją (cały kod prezentowany w tym podrozdziale jest zawarty w pliku
singletons.py).

@singleton
class Person:
onCall

    def __init__(self, name, hours, rate):
        # onCall pamięta Person
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

@singleton
class Spam:
onCall

    def __init__(self, val):
        # onCall pamięta Spam
        self.attr = val
bob = Person('Robert', 40, 10)
# Tak naprawdę wywołuje onCall
print(bob.name, bob.pay())
anna = Person('Anna', 50, 20)
# Ten sam pojedynczy obiekt
print(anna.name, anna.pay())
X = Spam(42)
# Jeden obiekt Person, jeden
objekt Spam
Y = Spam(99)
print(X.attr, Y.attr)

```

Gdy klasa Person bądź Spam zostaje później wykorzystana do utworzenia instancji, warstwa logiki opakowującej udostępniana przez dekorator przekierowuje wywołania tworzące instancję do funkcji onCall, która z kolei wywołuje funkcję getInstance zarządzającą pojedynczą instancją na klasę i ją współdzielącą, bez względu na to, ile wywołań tworzących zostanie wykonanych. Oto wynik powyższego kodu:

```
c:\code> python singletons.py
Robert 400
Robert 400
```

Alternatywne rozwiązania

Co ciekawe, możemy także wykorzystać rozwiązanie alternatywne, jeśli jesteśmy w stanie użyć instrukcji `nonlocal` (dostępnej w Pythonie 3.x) do modyfikacji zmiennych z zakresu zawierającego, zgodnie z wcześniejszym opisem. Poniższa alternatywa daje identyczny wynik, wykorzystując tylko jeden *zakres funkcji zawierającej* na klasę zamiast jednego wpisu do globalnej tabeli na klasę. Poniższa wersja działa tak samo, jednak nie jest uzależniona od nazw z zakresu globalnego poza dekoratorem. (Należy zwrócić uwagę, że w instrukcji warunkowej można użyć operatora `is` zamiast `==`; niemniej jednak jest to trywialne porównanie).

```
# Tylko w wersji 3.x: nonlocal

def singleton(aClass):                      # W momencie dekoracji @
    instance = None
    def onCall(*args):                      # W momencie tworzenia instancji
        nonlocal instance                  # Instrukcja nonlocal z wersji 3.x
        if instance == None:
            instance = aClass(*args)      # Jeden zakres na klasę
        return instance
    return onCall
```

W Pythonie 3.x oraz 2.x możemy także napisać takie samodzielne rozwiązanie za pomocą atrybutów funkcji lub klasy. W pierwszym z poniższych kodów wykorzystana jest pierwsza opcja. Wykorzystany jest fakt, że dekorowana jest tylko jedna *funkcja onCall*. Przestrzeń nazw obiektów pełni tutaj tę samą rolę, co otaczający zakres. W drugim kodzie do dekorowania wykorzystywana jest jedna *instancja*, a nie obejmujący zakres czy globalna tabela. Tak naprawdę wykorzystywany jest tu ten sam wzorzec projektowy, który później zobaczymy w często spotykanym rozwiążaniu związanym z dekoratorem klasy. Tutaj *potrzebna* jest nam tylko jedna instancja, jednak nie zawsze tak jest.

```
# Wersje 3.x i 2.x: atrybuty funkcji, klasy (alternatywne kodowanie)

def singleton(aClass):                      # W momencie dekoracji @
    def onCall(*args, **kwargs):              # W momencie tworzenia
        instance
        if onCall.instance == None:
            onCall.instance = aClass(*args, **kwargs)  # Jedna funkcja na klasę
        return onCall.instance
    onCall.instance = None
    return onCall

class singleton:
    def __init__(self, aClass):              # W momencie dekoracji @
        self.aClass = aClass
        self.instance = None
```

```

    def __call__(self, *args, **kwargs):
        # W momencie tworzenia
        instancji

        if self.instance == None:
            self.instance = self.aClass(*args, **kwargs) # Jedna instancja na
            klasę

        return self.instance

```

By uczynić z tego dekoratora prawdziwe narzędzie ogólnego przeznaczenia, należy przechować go w pliku modułu, który można importować, a także dokonać indentacji kodu testu samosprawdzającego pod sprawdzeniem `_name_` (pozostawię to jako sugerowane ćwiczenie). Drugie rozwiązanie oparte na klasie jest niezależne od wersji, a jego struktura jest lepiej przygotowania do rozwijania w przyszłości.

Śledzenie interfejsów obiektów

Przykład z singletonem z poprzedniego podrozdziału ilustrował użycie dekoratorów klas do zarządzania *wszystkimi* instancjami klasy. Kolejny często stosowany przypadek użycia dekoratorów klas rozszerza interfejs *każdej* z wygenerowanych instancji. Dekoratory klas mogą właściwie instalować na instancjach warstwę logiki opakowującej zarządzającą w jakiś sposób dostępem do ich interfejsów.

Przykładowo w rozdziale 31. metoda przeciążania operatora `__getattr__` zaprezentowana jest jako sposób opakowania całych interfejsów obiektów osadzonych instancji w celu zaimplementowania wzorca projektowego *delegacji*. Podobne przykłady widzieliśmy również w omówieniu zarządzanych atrybutów w poprzednim rozdziale. Warto przypomnieć, że metoda `__getattr__` jest wykonywana przy pobraniu niezdefiniowanej nazwy atrybutu. Ten punkt zaczepienia możemy wykorzystać do przechwytywania wywołań metod w klasie kontrolera i przekazywania ich do osadzonego obiektu.

Jako punkt odniesienia poniżej znajduje się oryginalny przykład delegacji niezawierający dekoratora, działający na dwóch obiektach typów wbudowanych.

```

class Wrapper:

    def __init__(self, object):
        self.wrapped = object # Zapisanie obiektu

    def __getattr__(self, attrname):
        print('Śledzenie:', attrname) # Śledzenie pobrania
        return getattr(self.wrapped, attrname) # Delegacja pobrania

    >>> x = Wrapper([1,2,3]) # Opakowanie listy
    >>> x.append(4) # Wydelegowanie do metody
    listy
    Śledzenie: append
    >>> x.wrapped # Wyświetlenie mojej
    składowej
    [1, 2, 3, 4]
    >>> x = Wrapper({"a": 1, "b": 2}) # Opakowanie słownika

```

```

>>> list(x.keys())
# Wydelegowanie do metody
słownika

Śledzenie: keys
użyć list()

['a', 'b']

```

W powyższym kodzie klasa `Wrapper` przechwytuje próby dostępu do każdego z atrybutów opakowanego obiektu, wyświetla komunikat śledzenia i wykorzystuje funkcję wbudowaną `getattr` do przekazania żądania do opakowanego obiektu. W szczególności śledzi ona próby dostępu do atrybutów wykonywane *poza* klasą opakowanego obiektu. Próby dostępu wewnętrz metod opakowanego obiektu nie są przechwytywane i są normalnie wykonywane. Cały ten model interfejsu różni się od zachowania dekoratorów funkcji, które opakowują tylko jedną, określona metodę.

Śledzenie interfejsów za pomocą dekoratorów klas

Dekoratory klas udostępniają alternatywny i wygodny sposób zapisu w kodzie techniki `__getattribute__` służącej do opakowania całego interfejsu. W Pythonie 2.x oraz 3.x poprzedni przykład z klasą można zapisać w postaci dekoratora klasy uruchamiającego tworzenie opakowanej instancji, zamiast przekazywać przygotowaną wcześniej instancję do konstruktora obiektu opakowującego (rozszerzonego tutaj również w celu obsługiwanego argumentów ze słowami kluczowymi w `**kwargs`, a także zliczania liczby wykonanych prób dostępu).

```

def Tracer(aClass):                                # W momencie dekoracji @
    class Wrapper:
        def __init__(self, *args, **kwargs):      # W momencie tworzenia
            self.fetches = 0
            self.wrapped = aClass(*args, **kwargs)  # Użycie nazwy z zakresu
            # funkcji zawierającej

        def __getattribute__(self, attrname):
            print('Śledzenie: ' + attrname)          # Przechwytuje wszystko
            # oprócz własnych atrybutów
            self.fetches += 1
            return getattr(self.wrapped, attrname)  # Delegacja do opakowanego
            # obiektu

        return Wrapper
    if __name__ == '__main__':
        @Tracer
        class Spam:                               # Spam = Tracer(Spam)
            def display(self):                   # Klasa Spam ponownie
                dowiązana do Wrapper
                print('Mielonka!' * 8)

        @Tracer
        class Person:                            # Person = Tracer(Person)
            def __init__(self, name, hours, rate): # Wrapper pamięta Person

```

```
    self.name = name
    self.hours = hours
    self.rate = rate

def pay(self):
    # Próby dostępu spoza klasy
    # są śledzone

    return self.hours * self.rate
    # Próby dostępu z wewnętrz
metody nie są śledzone

    # Wywołuje Wrapper()
    # Wywołuje __getattr__

food = Spam()
food.display()
print([food.fetches])

    # Wywołuje __getattribute__()

bob = Person('Robert', 40, 50)
    # Obiekt bob jest tak
naprawdę instancją Wrapper
    # Wrapper osadza instancje

    print(bob.name)
Person
    # Obiekt bob ma inny stan

    print(bob.pay())
    # Wywołuje __call__()

    print('')
    # Wywołuje __str__()

anna = Person('Anna', rate=100, hours=60)
    # Obiekt anna jest inną
instancją Wrapper
    # z inną klasą Person

    print(anna.name)
    print(anna.pay())
    # Wywołuje __call__()

    print(bob.name)
    print(bob.pay())
    # Obiekt bob ma inny stan

    print([bob.fetches, anna.fetches])
    # Atrybuty klasy Wrapper nie
są śledzone
```

Istotne jest, by zwrócić uwagę na to, jak bardzo kod ten różni się od dekoratora śledzącego, z którym spotkaliśmy się wcześniej. W podrozdziale zatytułowanym „Kod dekoratorów funkcji” przyglądalismy się dekoratorom pozwalającym na śledzenie i pomiar czasu wywołań określonej funkcji lub metody. W przecieństwie do nich dzięki przechwytywaniu wywołań tworzących instancje dekorator klasy pozwala nam śledzić pełny interfejs obiektu — czyli próby dostępu do dowolnego z jego atrybutów.

Poniżej znajdują się dane wyjściowe zwracane przez powyższy kod w wersjach 2.x oraz 3.x. Próby pobrania atrybutów instancji zarówno klasy `Spam`, jak i `Person` wywołują logikę metody `__getattr__` klasy `Wrapper`, ponieważ obiekty `food` oraz `bob` są tak naprawdęinstancjami klasy `Wrapper` dzięki przekierowaniu wywołań tworzących instancje przez dekorator.

```
c:\code> python interfacetracer.py
```

Śledzenie: display

Mielonka! Mielonka! Mielonka! Mielonka! Mielonka! Mielonka! Mielonka! Mielonka!

[1]

Śledzenie: name

Robert

```
Śledzenie: pay
2000
Śledzenie: name
Anna
Śledzenie: pay
6000
Śledzenie: name
Robert
Śledzenie: pay
2000
[4, 2]
```

Należy zwrócić uwagę, że w jednej dekoracji wykorzystywana jest jedna klasa `Wrapper` z zachowaniem stanu, generowana przez zagnieźdzoną instrukcję `class` w funkcji `Tracer`. Ponadto każda instancja uzyskuje własny licznik odczytów dzięki generowaniu nowej instancji klasy `Wrapper`. Jak się przekonamy, tego rodzaju kodowanie jest bardziej skomplikowane niż na pożór wydaje.

Stosowanie dekoratorów klas z wbudowanymi typami

Warto zauważyć, że poprzedni kod dekoruje klasę zdefiniowaną przez użytkownika. Tak jak w oryginalnym przykładzie z rozdziału 31., możemy także wykorzystać dekorator do opakowania typu wbudowanego, takiego jak lista, o ile albo wykorzystamy klasę podrzędną w celu umożliwienia składni dekoracji, albo dekorację wykonamy ręcznie — składnia dekoratora wymaga instrukcji `class` dla wiersza ze znakiem `@`. W poniższym kodzie zmienna `x` jest tak naprawdę znowu instancją klasy `Wrapper` z powodu pośredniego działania dekoracji:

```
>>> from tracer import Tracer
>>> @Tracer
... class MyList(list): pass # MyList = Tracer(MyList)
>>> x = MyList([1, 2, 3]) # Wywołuje Wrapper()
>>> x.append(4) # Wywołuje __getattr__, append
Śledzenie: append
>>> x.wrapped
[1, 2, 3, 4]
>>> WrapList = Tracer(list) # Lub ręczne wykonanie dekoracji
>>> x = WrapList([4, 5, 6]) # Inaczej wymagana instrukcja
klasy podrzędnjej
>>> x.append(7)
Śledzenie: append
>>> x.wrapped
[4, 5, 6, 7]
```

Rozwiążanie z dekoratorem pozwala nam na przeniesienie tworzenia instancji do samego dekoratora zamiast wymagania przekazania utworzonego wcześniej obiektu. Choć różnica wydaje się nieznaczna, pozwala nam to na zachowanie normalnej składni tworzenia instancji i ogólnie na skorzystanie ze wszystkich zalet dekoratorów. Zamiast wymagać, by wszystkie wywołania tworzące instancję ręcznie przekierowywały obiekty do obiektu pośredniczącego, wystarczy, że rozszerzymy klasy za pomocą składni dekoratorów.

```
@Tracer                                # Rozwiążanie z dekoratorem

class Person: ...

bob = Person('Robert', 40, 50)
anna = Person('Anna', rate=100, hours=60)

class Person: ...                          # Rozwiążanie bez dekoratora

bob = Wrapper(Person('Robert', 40, 50))
anna = Wrapper(Person('Anna', rate=100, hours=60))
```

Zakładając, że będziemy tworzyć więcej niż jedną instancję klasy, dekoratory zazwyczaj będą lepszym rozwiązaniem z punktu widzenia zarówno wielkości kodu, jak i jego późniejszego utrzymywania.



Uwaga na temat wersji: Opisany dekorator śledzący działa poprawnie z atrybutami o jawnych nazwach we wszystkich wersjach Pythona. Jak wiemy z rozdziału 38., 32. i kilku jeszcze miejsc, metoda `__getattribute__` przechwytuje próby dostępu do metod przeciążających operatory, takich jak `__str__` czy `__repr__`, w typowych klasach w Pythonie 2.x, ale nie w klasach w nowym stylu wersji 3.x.

W Pythonie 3.x instancje klas dziedziczą wartości domyślne dla części (jednak nie wszystkich) z tych nazw po klasie (a tak naprawdę po automatycznej klasie nadzędnej `object`). Co więcej, w wersji 3.x atrybuty wywoływane w sposób niejawny dla operacji wbudowanych, takich jak wyświetlanie czy `+`, nie są przekierowywane za pośrednictwem metody `__getattribute__` (bądź jej kremnego, `__getattribute__`). Klasy w nowym stylu wyszukują takie metody w klasach i całkowicie pomijają normalne wyszukiwanie w instancjach.

Tutaj oznacza to, że obiekt opakowujący oparty na metodzie `__getattribute__` automatycznie będzie śledził i przekazywał wywołania przeciążające operatory w Pythonie 2.x, ale w wersji 3.x już nie. By się o tym przekonać, wystarczy wyświetlić zmienną `x` w sposób bezpośredni na końcu poprzedniej sesji interaktywnej. W Pythonie 2.x metoda `__repr__` atrybutu jest śledzona i lista wyświetlana jest w oczekiwany sposób, natomiast w wersji 3.x śledzenie nie występuje, a lista wyświetlana jest z wykorzystaniem domyślnego sposobu wyświetlania dla klasy `Wrapper`.

```
>>> x                                # Python 2.x
Śledzenie: __repr__
[4, 5, 6, 7]
>>> x                                # Python 3.x
<interfacetracer.Tracer.<locals>.Wrapper object at 0x02946358>
```

By kod działał w ten sam sposób w Pythonie 3.x, należy ponownie zdefiniować metody przeciążania operatorów w klasie opakowującej — ręcznie, za pomocą narzędzi lub definicji w klasie nadzędnej. Z tą różnicą między wersjami spotkamy się ponownie w dekoratorze `Private` w dalszej części rozdziału,

gdzie będą opisane sposoby dodawania metod wymaganych w takim kodzie w wersji 3.x.

Uwagi na temat klas II — zachowanie większej liczby instancji

Co ciekawe, funkcję dekoratora z tego przykładu można by *prawie* zapisać w kodzie jako klasę, a nie funkcję, z odpowiednim protokołem przeciążania operatora. Poniższa, nieco uproszczona alternatywa działa w podobny sposób, ponieważ jej metoda `__init__` wywoływana jest, kiedy dekorator `@` zostanie zastosowany do klasy, natomiast jej metoda `__call__` uruchamiana jest, kiedy tworzona jest instancja klasy podmiotowej. Nasze obiekty są tym razem tak naprawdę instancjami klasy `Tracer` i w gruncie rzeczy wymieniamy tutaj referencję do zakresu funkcji zawierającej na atrybut instancji.

```
class Tracer:

    def __init__(self, aClass):                      # W momencie dekoracji @
        self.aClass = aClass                         # Użycie atrybutu instancji

    def __call__(self, *args):                        # W momencie tworzenia
        instancji

        self.wrapped = self.aClass(*args)            # JEDNA (OSTATNIA) INSTANCJA
        NA KLASĘ!

        return self

    def __getattr__(self, attrname):
        print('Śledzenie: ' + attrname)
        return getattr(self.wrapped, attrname)

@Tracer                                         # Wywołuje __init__
class Spam:                                       # Jak: Spam = Tracer(Spam)

    def display(self):
        print('Mielonka!' * 8)

    ...

food = Spam()                                     # Wywołuje __call__
food.display()                                    # Wywołuje __getattr__
```

Jak jednak widzieliśmy wcześniej, alternatywa z klasą obsługuje większą liczbę klas, podobnie jak poprzednie rozwiązanie, jednak w zasadzie nie działa dla *większej liczby instancji* określonej klasy — każde wywołanie tworzące instancję wywołuje metodę `__call__`, która nadpisuje poprzednią instancję. W rezultacie klasa `Tracer` zapisuje tylko jedną instancję — ostatnią utworzoną. Warto samodzielnie poeksperymentować, by się o tym przekonać, jednak poniżej znajduje się przykład tego problemu.

```
@Tracer
class Person:                                     # Person = Tracer(Person)

    def __init__(self, name):                     # Klasa Wrapper dowiązana do
        Person
```

```
    self.name = name
bob = Person('Robert')                      # Obiekt bob jest tak naprawdę
instancją Wrapper
print(bob.name)                            # Wrapper osadza Person
Anna = Person('Anna')
print(anna.name)                            # Obiekt anna nadpisuje obiekt bob
print(bob.name)                            # UPS: teraz obiekt bob nazywa się
'Anna'!
```

Wynik powyższego kodu będzie następujący. Ponieważ ta klasa śledząca ma tylko jedną współdzieloną instancję, druga z nich nadpisuje pierwszą.

Śledzenie: name

Robert

Śledzenie: name

Anna

Śledzenie: name

Anna

Problemem jest tutaj zle *zachowanie stanu*. Tworzymy jedną instancję dekoratora na klasę, ale nie na instancję klasy, przez co jedynie ostatnia instancja zostanie zachowana. Rozwiązaniem, jak w poprzednich rozwiązańach na temat klas w przypadku dekoracji metod, jest porzucenie dekoratorów opartych na klasach.

Wersja klasy `Tracer` oparta na funkcji `działa` dla większej liczby instancji, ponieważ każde wywołanie tworząceinstancję tworzy nową instancję klasy `Wrapper`, zamiast nadpisywać stan pojedynczej, współdzielonej instancji `Tracer`. Oryginalna wersja bez dekoratora również poprawnie obsługuje większą liczbę instancji — z tych samych przyczyn. Dekoratory są nie tylko nieco magiczne, ale także dość subtelne w swoim działaniu!

Dekoratory a funkcje zarządzające

Bez względu na subtelności tego typu przykład dekoratora klasy `Tracer` nadal opiera się na metodzie `__getattr__` przechwytyjącej pobrania opakowanego i osadzonego obiektu instancji. Jak widzieliśmy wcześniej, jedyne, co udało nam się osiągnąć, to przeniesienie wywołania tworzącegoinstancję wewnątrz klasy zamiast przekazywania instancji do funkcji zarządzającej. W przypadku oryginalnego przykładu śledzenia bez dekoratora po prostu w inny sposób zapisaliśmy kod tworzenia instancji.

```
food = Spam()                      # Normalna składnia tworzenia
instancji
```

Dekoratory klasy przesuwają wymagania w zakresie specjalnej składni z wywołania tworzącego instancję do samej instrukcji `class`. Tak samo było w przypadku przykładu singletona umieszczonego wcześniej — zamiast dekorowania klasy i użycia normalnych wywołań tworzących instancję po prostu przekazywaliśmy klasę oraz jej argumenty konstrukcyjne do funkcji zarządzającej.

```
instances = {}

def getInstance(aClass, *args, **kwargs):
    if aClass not in instances:
        instances[aClass] = aClass(*args, **kwargs)
    return instances[aClass]

bob = getInstance(Person, 'Robert', 40, 10)      # Nie: bob =
Person('Robert', 40, 10)
```

Alternatywnie moglibyśmy wykorzystać możliwości Pythona w zakresie introspekcji do pobrania klasy z utworzonej już instancji (zakładając, że tworzenie początkowej instancji jest dopuszczalne).

```
instances = {}

def getInstance(object):
    aClass = object.__class__
    if aClass not in instances:
        instances[aClass] = object
    return instances[aClass]

bob = getInstance(Person('Robert', 40, 10))      # Nie: bob =
Person('Robert', 40, 10)
```

Tak samo jest w przypadku *dekoratorów funkcji*, takich jak napisana wcześniej funkcja śledząca. Zamiast dekorować funkcję logiką przechwytyującą późniejsze wywołania, moglibyśmy po prostu przekazać funkcję oraz jej argumenty do funkcji zarządzającej przesyłającej wywołanie.

```
def func(x, y):                      # Wersja bez dekoratora
    ...
func(*args)                         # def tracer(func, args): ...

result = tracer(func, (1, 2))       # Specjalna składnia wywołania
@tracer

def func(x, y):                      # Wersja z dekoratorem
    ...
= tracer(func)                     # Ponownie dowiązuje nazwę: func
result = func(1, 2)                 # Normalna składnia wywołania
```

Rozwiązania z funkcjami zarządzającymi, jak powyższe, nie tylko przenoszą ciężar użycia specjalnej składni na *wywołania*, zamiast oczekiwania składni dekoracji w definicjach funkcji oraz klas, ale też pozwalają wybiórczo modyfikować poszczególne wywołania.

Do czego służą dekoratory (raz jeszcze)

Dlaczego zatem zaprezentowałem właśnie sposoby *nieużywania* dekoratorów w implementacji klas typu singleton? Jak wspomniałem na początku niniejszego rozdziału, dekoratory często oznaczają pewne kompromisy. Choć składnia ma znaczenie, zbyt często zapominamy zadać pytanie „po co?”, kiedy spotykamy się z nowymi narzędziami. Skoro już widzieliśmy, jak działają dekoratory, spróbujmy poświęcić chwilę na spojrzenie na nie z pewnej perspektywy.

Jak większość funkcji języka, dekoratory mają zarówno wady, jak i zalety. Przykładowo wśród wad możemy wymienić dwa potencjalne niedociągnięcia dekoratorów klas:

Zmiany typu

Jak widzieliśmy, kiedy wstawiane są obiekty opakowujące, udekorowana funkcja bądź klasa nie zachowuje *oryginalnego typu*. Nazwa dowiązywana jest ponownie do obiektu opakowującego, co może mieć znaczenie w programach wykorzystujących nazwy obiektów lub sprawdzających ich typy. W przykładzie z singletonem zarówno rozwiązanie z dekoratorem, jak i funkcją zarządzającą zachowuje oryginalny typ klasy dla instancji. W kodzie śledzącym żadne z rozwiązań tego nie robi, ponieważ wymagane są obiekty opakowujące. Oczywiście w polimorficznym języku, takim jak Python, należy unikać sprawdzania typów, ale od większości reguł są wyjątki.

Dodatkowe wywołania

Warstwa opakowująca dodana przez dekorację powoduje dodatkowe obciążenie w zakresie wydajności spowodowane przez *dodatkowe wywołania* wykonywane za każdym razem, gdy wywoływany jest udekorowany obiekt. Wywołania są operacjami pochłaniającymi stosunkowo dużo czasu, przez co dekorujące obiekty opakowujące mogą spowolnić program. W kodzie śledzącym oba rozwiązania wymagają, by każdy atrybut był przekierowywany za pośrednictwem warstwy opakowującej. Przykład singletona unika dodatkowych wywołań, zachowując oryginalny typ klasy.

Wszystko lub nic

Dekorator modyfikuje funkcję lub klasę i wpływa na wszystkie późniejsze wywołania udekorowanego obiektu. Dzięki temu obiekt jest wprawdzie wykorzystywany w ujednolicony sposób, ale czasami jest to przeszkodą, jeżeli modyfikacja ma dotyczyć wybranych wywołań.

Żadna z poruszonych kwestii nie jest jednak zbyt poważna. W przypadku większości programów różnica typu raczej nie będzie miała znaczenia, a utrata szybkości związana z dodatkowymi wywołaniami będzie nieznaczna. Co więcej, ta ostatnia występuje jedynie wtedy, gdy wykorzystywane są obiekty opakowujące, i często można sobie z nią poradzić, usuwając dekorator, gdy wymagana jest optymalna wydajność. Utrata szybkości występuje również w rozwiązańach bez dekoratora, które dodają logikę opakowującą (w tym w *metaklasach*, o czym przekonamy się w rozdziale 40.).

I odważnie, jak widzieliśmy na początku rozdziału, dekoratory mają trzy istotne zalety. W porównaniu z rozwiązaniami z funkcją zarządzającą (inaczej: pomocniczą) z poprzedniego podrozdziału dekoratory to także:

Jasna, jawnia składnia

Dekoratory sprawiają, że rozszerzenie jest jasne i oczywiste. Ich składnia ze znakiem @ jest łatwiejsza do rozpoznania od specjalnego kodu wywołań, który może się pojawić w dowolnym miejscu pliku źródłowego. W naszych przykładach singletona i śledzenia wiersze z dekoratorem wydają się łatwiejsze do zauważenia niż dodatkowy kod w wywołaniach. Co więcej, dekoratory pozwalają na wykorzystywanie normalnej, znanej wszystkim programistom Pythona składni funkcji oraz wywołań tworzących instancje.

Utrzymywanie kodu

Dekoratory pozwalają uniknąć powtarzania kodu rozszerzającego w każdej funkcji czy wywołaniu klasy. Ponieważ pojawiają się tylko raz, w samej definicji klasy lub funkcji, ograniczają powtarzalność kodu i upraszczają utrzymywanie go w przyszłości. W przypadku naszego kodu singletona oraz śledzenia musielibyśmy zastosować specjalny kod w każdym wywołaniu, by skorzystać z rozwiązania z funkcją zarządzającą. Dodatkowa praca wymagana była zatem nie tylko na początku, ale także w przypadku wszelkich modyfikacji, które będą musiały być wykonane w przyszłości.

Spójność

Dekoratory sprawiają, że mniej prawdopodobne będzie to, iż programista zapomni skorzystać z wymaganej logiki opakowującej. Wynika to w dużej mierze z dwóch poprzednich zalet. Ponieważ dekoracja jest jawną i pojawia się tylko raz, w samych udekorowanych obiektach, dekoratory promują bardziej spójne i jednolite wykorzystywanie API w stosunku do specjalnego kodu, który należy umieszczać w każdym wywołaniu. W przykładzie z singletonem łatwo byłoby zapomnieć o przekierowaniu wszystkich wywołań tworzących klasę do specjalnego kodu, co całkowicie zaprzeczyłoby idei zarządzania singletonem.

Dekoratory promują także *hermetyzację* kodu w celu ograniczenia jego powtarzalności i zminimalizowania wysiłku wymaganego przy jego utrzymywaniu w przyszłości. Rozszerzający kod trzeba wypisać tylko w jednym miejscu, w funkcji dekoratora, i nie trzeba go powielać w każdym miejscu, w którym ma być użyty. Choć inne narzędzia strukturyzujące kod także to robią, w przypadku dekoratorów jest to naturalne dla wszystkich zadań związanych z rozszerzaniem.

Żadna z wymienionych zalet nie nakazuje jednak bezwzględnie używać składni dekoratorów, a samo ich wykorzystanie jest wyborem stylistycznym. Jednak większość programistów uznaje dekoratory za rozwiązanie korzystniejsze, zwłaszcza w roli narzędzi umożliwiających poprawne korzystanie z bibliotek oraz API.



Anegdota historyczna: Pamiętam podobną argumentację za i przeciw funkcjom *konstruktora* w klasach. Przed wprowadzeniem metod `__init__` ten sam efekt można było często osiągnąć po ręcznym wywołaniu metody na instancji przy tworzeniu jej (na przykład `X=Class().init()`). Z czasem jednak, choć był to wybór w dużej mierze stylistyczny, składnia `__init__` stała się preferowana, gdyż była bardziej oczywista, jawną, spójną i łatwiejszą w utrzymaniu. Choć każdy powinien decydować za siebie, dekoratory zdają się kłaść na szali te same zalety.

Bezpośrednie zarządzanie funkcjami oraz klasami

Większość z naszych przykładów z niniejszego rozdziału została zaprojektowana pod kątem przechwytywania wywołań funkcji oraz wywołań tworzących instancje. Choć jest to typowe zastosowanie dekoratorów, nie są one ograniczone do tej roli. Ponieważ dekoratory działają, przekazując nowe funkcje i klasy do kodu dekoratora, można je także wykorzystać do zarządzania samymi obiektami funkcji oraz klas, a nie tylko wykonywanymi do nich wywołaniami.

Wyobraźmy sobie na przykład, że metody bądź klasy wykorzystywane przez aplikację muszą być rejestrowane w API w celu późniejszego przetworzenia (być może API wywoła te obiekty później, w odpowiedzi na zdarzenia). Choć można udostępnić funkcję rejestrującą, którą

wywołamy ręcznie po zdefiniowaniu obiektów, dekoratory uczynią nasze intencje bardziej oczywistymi.

Poniższa prosta implementacja tego pomysłu definiuje dekorator, który można zastosować zarówno do funkcji, jak i klas w celu dodania obiektu do rejestru opartego na słowniku. Ponieważ zwraca ona sam obiekt zamiast obiektu opakowującego, nie przechwytuje późniejszych wywołań.

```
# Rejestrowanie udekorowanych obiektów w API

from __future__ import print_function    # 2.x
registry = {}

def register(obj):                      # Dekorator zarówno klasy, jak i
funkcji

    registry[obj.__name__] = obj          # Dodanie do rejestru

    return obj                           # Zwrócenie samego obiektu, a
nie obiektu opakowującego

@register

def spam(x):

    return(x ** 2)                      # spam = register(spam)

@register

def ham(x):

    return(x ** 3)

@register

class Eggs:                            # Eggs = register(Eggs)

    def __init__(self, x):
        self.data = x ** 4

    def __str__(self):
        return str(self.data)

print('Rejestr:')

for name in registry:

    print(name, '=>', registry[name], type(registry[name]))

print('\nWywołania ręczne:')

print(spam(2))                         # Ręczne wywołanie obiektów

print(ham(2))                          # Późniejsze wywołania nie są
przechwytywane

X = Eggs(2)

print(X)

print('\nWywołania z rejestru:')

for name in registry:
```

```
print(name, '=>', registry[name](3))      # Wywołanie z rejestru
```

Po wykonaniu powyższego kodu udekorowane obiekty dodawane są do rejestru po nazwach, jednak przy późniejszym wywołaniu nadal działają w taki sposób, jak je zaprojektowano, bez przekierowania do warstwy opakowującej. Tak naprawdę nasze obiekty można wykonać zarówno ręcznie, jak i z wewnętrz tabeli rejestru.

```
c:\code> py -3 registry-deco.py
```

Rejestr:

```
spam => <function spam at 0x02969158> <class 'function'>
ham => <function ham at 0x02969400> <class 'function'>
Eggs => <class '__main__.Eggs'> <class 'type'>
```

Wywołania ręczne:

```
4
8
16
```

Wywołania z rejestru:

```
Eggs => 81
ham => 27
spam => 9
```

Technikę tę mogliby wykorzystywać na przykład interfejs użytkownika do rejestrowania programów obsługi wywołań zwrotnych dla działań użytkownika. Programy obsługi mogą być rejestrowane za pomocą nazwy funkcji bądź klasy, jak powyżej; można także wykorzystać argumenty dekoratora do określenia podmiotowego zdarzenia. Dodatkowa instrukcja `def` zawierająca dekorator mogłaby zostać użyta do zachowania takich argumentów w celu zastosowania ich w dekoracji.

Powyższy przykład jest dość sztuczny, ale technika ta jest bardzo uniwersalna. Dekoratory funkcji można na przykład wykorzystać także do przetwarzania atrybutów funkcji, natomiast dekoratory klas mogą w sposób dynamiczny wstawiać nowe atrybuty klas czy nawet nowe metody. Rozważmy poniższe dekoratory funkcji. Przypisuję one atrybuty funkcji do informacji rekordów w celu późniejszego wykorzystania ich przez API, jednak nie wstawiają warstwy opakowującej, przechwytyjącej późniejsze wywołania.

```
# Bezpośrednie rozszerzenie udekorowanych obiektów
>>> def decorate(func):
...     func.marked = True                      # Przypisanie atrybutu funkcji dla
późniejszego użycia
...     return func
...
>>> @decorate
... def spam(a, b):
...     return a + b
...
```

```

>>> spam.marked
True
>>> def annotate(text):                      # To samo, jednak wartość jest
    argumentem dekoratora
    ...
    def decorate(func):
        ...
        func.label = text
        ...
        return func
    ...
    return decorate
    ...

>>> @annotate('mielonka dane')
... def spam(a, b):                         # spam = annotate(...)(spam)
...     return a + b
...
>>> spam(1, 2), spam.label
(3, 'mielonka dane')

```

Dekoratory tego typu rozszerzają funkcje oraz klasy w sposób bezpośredni, bez przechwytywania ich późniejszych wywołań. Więcej przykładów dekoracji klas zarządzającej bezpośrednio klasami zobaczymy w kolejnym rozdziale, ponieważ kwestia ta okazuje się pokrywać z dziedziną *metaklas*. W dalszej części niniejszego rozdziału zajmiemy się działaniem dwóch większych przypadków użycia dekoratorów.

Przykład — atrybuty „prywatne” i „publiczne”

Ostatnie dwie części niniejszego rozdziału prezentują większe przykłady zastosowania dekoratorów. Oba zawierają skróconą część opisową — po części dlatego, że rozdział ten już przekroczył przewidziany dla niego limit długości, ale również z powodu tego, że Czytelnik powinien już na tyle dobrze rozumieć podstawy dekoratorów, by potrafić przestudiować przykłady samodzielnie. Są to narzędzia ogólnego zastosowania i dają nam możliwość przekonania się, w jaki sposób koncepcje związane z dekoratorami łączą się ze sobą w bardziej użytecznym kodzie.

Implementacja atrybutów prywatnych

Poniższy *dekorator klasy* implementuje deklarację atrybutów instancji klasy jako prywatnych. Oznacza to, że atrybuty są przechowywane w instancji lub dziedziczone po jednej z jej klas. Nie pozwala on na pobieranie i modyfikację takich atrybutów *spoza* udekorowanej klasy, jednak nadal pozwala samej klasie na swobodny dostęp do tych zmiennych wewnętrz jej metod. Kod ten nie do końca jest tym samym co mechanizmy języków C++ czy Java, ale umożliwia podobną kontrolę dostępu jako element opcjonalny Pythona.

W rozdziale 30. widzieliśmy już niepełną, wstępna implementację prywatności atrybutów instancji w przypadku *modyfikacji*. Wersja z tego rozdziału rozszerza tę koncepcję w taki

sposób, by sprawdzać również próby *pobierania* atrybutu. Do implementacji tego modelu wykorzystuje także delegację zamiast dziedziczenia. Tak naprawdę w pewnym sensie jest to tylko rozszerzenie dekoratora klasy śledzącej atrybuty, z którym spotkaliśmy się wcześniej.

Choć przykład ten wykorzystuje do implementacji prywatności atrybutów nowość składniową, jaką są dekoratory klas, przechwytywanie atrybutów jest w nim w dużej mierze nadal oparte na metodach przeciążania operatorów `_getattr_` oraz `_setattr_`, z którymi spotkaliśmy się w poprzednich rozdziałach. Kiedy wykryta zostaje próba dostępu do atrybutu prywatnego, ta wersja kodu wykorzystuje instrukcję `raise` do zgłoszenia wyjątku wraz z komunikatem o błędzie. Wyjątek ten można przechwycić w instrukcji `try` lub pozwolić mu na zakończenie skryptu.

Poniżej znajduje się kod przykładu wraz z testem samosprawdzającym zamieszczonym na dole pliku. Działa on zarówno w Pythonie 2.x, jak i 3.x, ponieważ wykorzystuje składnię `print` i `raise` stosowaną w obu wersjach, jednak przechwytuje atrybuty metod przeciążania operatorów jedynie w wersji 2.x (więcej na ten temat za chwile).

三

Plik access1.py (wersje 3.x i 2.x)

Prywatność dla atrybutów pobranych z instancji klas.

Przykłady użycia można znaleźć w kodzie testu samosprawdzającego na dole pliku.

Dekorator jest tym samym co: Doubler = Private('data', 'size')(Doubler).

Private zwraca onDecorator, onDecorator zwraca onInstance, a każda instancja onInstance osadza instancje Doubler.

三

```
traceMe = False

def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def Private(*privates):                      # privates w zakresie funkcji
zawierającej

    def onDecorator(aClass):                  # aClass w zakresie funkcji
zawierającej

        class onInstance:                   # Opakowane w atrybut instancji

            def __init__(self, *args, **kargs):
                self.wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):      # Moje atrybuty nie
wywołują metody getattr
                trace('pobranie:', attr)     # Inne zakładane w obiekcie
wrapped

                if attr in privates:
                    raise TypeError('pobranie atrybutu prywatnego: ' + attr)
                else:
                    return getattr(self.wrapped, attr)
```

```

        def __setattr__(self, attr, value):      # Próby dostępu z zewnątrz
            trace('ustawienie:', attr, value)    # Pozostałe działają
normalnie

        if attr == 'wrapped':                  # Pozwolenie na własne
            atrybuty
                self.__dict__[attr] = value     # Uniknięcie pętli
            elif attr in privates:
                raise TypeError('modyfikacja atrybutu prywatnego: ' + attr)
            else:
                setattr(self.wrapped, attr, value) # Opakowane atrybuty
obiektu

        return onInstance                      # Lub użycie __dict__

    return onDecorator

if __name__ == '__main__':
    traceMe = True

    @Private('data', 'size')                 # Doubler = Private(...)
(Doubler)

    class Doubler:
        def __init__(self, label, start):
            self.label = label               # Próby dostępu wewnętrz
podmiotowej klasy

            self.data = start                # Nie zostaje przechwycony:
wykonany normalnie

        def size(self):
            return len(self.data)          # Metody działają bez
sprawdzania

        def double(self):                 # Ponieważ prywatność nie
jest dziedziczona
            for i in range(self.size()):
                self.data[i] = self.data[i] * 2

        def display(self):
            print('%s => %s' % (self.label, self.data))

X = Doubler('X to', [1, 2, 3])
Y = Doubler('Y to', [-10, -20, -30])
# Poniższe testy kończą się powodzeniem

        print(X.label)                  # Próby dostępu spoza
podmiotowej klasy

```

```

X.display(); X.double(); X.display()                      # Przechwycony:
sprawdzony, wydelegowany

    print(Y.label)

    Y.display(); Y.double()

    Y.label = 'Mielonka'

    Y.display()

# Poniższe testy wszystkie kończą się niepowodzeniem (zgodnie z
zamierzeniami)

"""

print(X.size())                                         # Wyświetla "TypeError: pobranie
atrybutu prywatnego: size"

print(X.data)

X.data = [1, 1, 1]

X.size = lambda S: 0

print(Y.data)

print(Y.size())

"""

```

Kiedy `traceMe` zwraca `True`, kod testu samosprawdzającego pliku modułu zwraca następujące dane wyjściowe. Warto zwrócić uwagę na to, w jaki sposób dekorator przechwytuje i sprawdza zarówno próby pobierania, jak i przypisania wykonywane *poza* opakowaną klasą — jednak nie przechwytuje prób dostępu wykonywanych *wewnątrz* samej klasy.

```

c:\code> py -3 access1.py

[set: wrapped <__main__.Doubler object at 0x00000000029769B0>]
[set: wrapped <__main__.Doubler object at 0x00000000029769E8>]
[pobranie: label]

X to

[pobranie: display]
X to => [1, 2, 3]
[pobranie: double]
[pobranie: display]
X to => [2, 4, 6]
[pobranie: label]

Y to

[pobranie: display]
Y to => [-10, -20, -30]
[pobranie: double]
[ustawienie: label Mielonka]
```

```
[pobranie: display]  
Mielonka => [-20, -40, -60]
```

Szczegóły implementacji I

Powyższy kod jest nieco skomplikowany i najlepiej będzie prześledzić go samodzielnie w celu przekonania się, jak działa. By pomóc w tej analizie, poniżej znajduje się kilka elementów, na które warto zwrócić uwagę.

Dziedziczenie a delegacja

Pierwszy, wstępny przykład implementacji prywatności z rozdziału 30. wykorzystywał *dziedziczenie* do umieszanego metod `__setattr__` do przechwytywania prób dostępu. Dziedziczenie mocno to jednak utrudnia, ponieważ rozróżnienie pomiędzy dostępem z wewnętrz i z zewnątrz klasy nie jest tak oczywiste (dostęp z wewnętrz powinien móc się odbywać normalnie, natomiast dostęp z zewnątrz powinien być ograniczony). By to obejść, przykład z rozdziału 30. wymagał, aby dziedziczące klasy wykorzystywały przypisania `__dict__` w celu ustawnienia atrybutów — rozwiązań to można w najlepszym razie nazwać niepełnym.

Wersja zaprezentowana powyżej wykorzystuje *delegację* (osadzenie jednego obiektu wewnętrz innego) zamiast dziedziczenia. Taki wzorzec kodu lepiej nadaje się do naszego zadania, gdyż bardzo ułatwia rozróżnienie pomiędzy próbami dostępu z wewnętrz oraz z zewnątrz podmiotowej klasy. Dostęp do atrybutów spoza klasy podmiotowej jest przechwytywany przez metody przeciążania operatorów warstwy opakowującej i delegowany do klasy, jeśli jest poprawny. Próby dostępu wewnętrz samej klasy (na przykład za pośrednictwem `self` wewnętrz kodu metod) nie są przechwytywane i mogą być wykonywane normalnie bez sprawdzania, ponieważ prywatność nie jest tutaj dziedziczona.

Argumenty dekoratora

Wykorzystany tutaj dekorator klasy przyjmuje dowolną liczbę argumentów nazywających atrybuty prywatne. Tak naprawdę argumenty przekazywane są do funkcji `Private`, natomiast `Private` zwraca funkcję dekoratora, która ma być zastosowana do podmiotowej klasy. Argumenty wykorzystywane są zatem przed wystąpieniem dekoracji. Funkcja `Private` zwraca dekorator, który z kolei „pamięta” listę atrybutów prywatnych w postaci referencji do zakresu funkcji zawierającej.

Zachowywanie stanu i zakresy funkcji zawierającej

A skoro już mowa o zakresach funkcji zawierającej, tak naprawdę w powyższym kodzie zachowywanie stanu działa na *trzech poziomach*:

- Argumenty funkcji `Private` wykorzystywane są, zanim nastąpi dekoracja, i są zachowywane w postaci referencji do zakresu funkcji zawierającej — do wykorzystania w `onDecorator` oraz `onInstance`.
- Argument klasy dla `onDecorator` wykorzystywany jest w czasie dekoracji i zachowywany w postaci referencji do zakresu funkcji zawierającej — do wykorzystania w czasie tworzenia instancji.
- Obiekt opakowanej instancji zachowywany jest w postaci atrybutu instancji w `onInstance` — do wykorzystania, gdy w późniejszym czasie ma miejsce próba dostępu do atrybutów spoza klasy.

Wszystko to działa w miarę naturalnie w oparciu o reguły Pythona dotyczące zakresów oraz przestrzeni nazw.

Wykorzystanie `__dict__` oraz `__slots__` (i innych nazw wirtualnych)

Metoda `__setattr__` z powyższego kodu, próbując ustawić własny opakowany atrybut `onInstance`, opiera się na słowniku `_dict_` przestrzeni nazw atrybutów obiektu instancji. Jak wiemy z poprzedniego rozdziału, nie może ona przypisać atrybutu w sposób bezpośredni bez wykonania pętli. Wykorzystuje ona jednak funkcję wbudowaną `setattr` w miejsce `_dict_` w celu ustawienia atrybutów w samym opakowanym obiekcie. Co więcej, do pobrania atrybutów w opakowanym obiekcie wykorzystana zostaje metoda `getattr`, ponieważ mogą one być przechowywane w samym obiekcie bądź odziedziczone przez niego.

Z tej przyczyny kod ten będzie działał dla większości klas, w tym z „wirtualnymi” atrybutami opartymi na *slotach*, *właściwościach*, *deskryptorach*, a nawet metodzie `__getattr__` i jej podobnych. Klasa opakowująca, dzięki temu, że rezerwuje słownik przestrzeni nazw tylko dla siebie i wykorzystuje niezależne narzędzia, jest pozbawiona ograniczeń typowych dla innych narzędzi.

Niektóre osoby mogą pamiętać z rozdziału 32., że klasy w nowym stylu ze `__slots__` mogą nie przechowywać atrybutów w słowniku `_dict_` (a nawet mogą tych atrybutów nie mieć). Ponieważ jednak polegamy na `_dict_` jedynie na poziomie `onInstance`, a nie w opakowanej instancji, a także dlatego, że metody `setattr` oraz `getattr` mają zastosowanie do atrybutów opartych zarówno na `_dict_`, jak i na `__slots__`, nasz dekorator działa na klasach wykorzystujących dowolny z tych mechanizmów przechowywania. Z tego samego powodu dekoratory można stosować z właściwościami w nowym stylu oraz podobnymi narzędziami. Delegowane nazwy będą ponownie wyszukiwane w opakowanej instancji niezależnie od atrybutów samego pośrednika dekoratora.

Uogólnienie kodu pod kątem deklaracji atrybutów jako publicznych

Skoro już mamy implementację prywatności, dość łatwo jest uogólnić ten kod w celu dopuszczenia także deklaracji publicznych — są one właściwie odwrotnością deklaracji prywatnych, dlatego wystarczy, że będą negować wewnętrzny test. Przykład wymieniony w tym podrozdziale pozwala klasie wykorzystywać dekoratory do zdefiniowania zbioru publicznych lub prywatnych atrybutów instancji (atrbutów przechowywanych w instancji lub odziedziczonych po jej klasach) za pomocą następującej semantyki:

- `Private` deklaruje atrybuty instancji klas, których *nie można* pobierać lub przypisywać, z wyjątkiem pochodzących z wewnętrz kodu metod klasy. Oznacza to, że nie jest możliwy dostęp z zewnątrz do żadnej zmiennej zadeklarowanej jako prywatna, natomiast wszystkie zmienne niezadeklarowane w ten sposób można swobodnie pobierać i przypisywać z zewnątrz.
- `Public` deklaruje atrybuty instancji klas, które *można* pobierać lub przypisywać zarówno z zewnątrz klasy, jak i z wewnętrz jej metod. Oznacza to, że dostęp do każdej zmiennej zadeklarowanej jako publiczna jest możliwy z dowolnego miejsca, natomiast dostęp z zewnątrz do zmiennych niezadeklarowanych w ten sposób nie jest możliwy.

Deklaracje `Private` oraz `Public` mają się wzajemnie wykluczać. Kiedy używamy `Private`, wszystkie niezadeklarowane zmienne uznawane są za publiczne, natomiast jeśli użyjemy `Public`, wszystkie niezadeklarowane zmienne uznawane są za prywatne. Są one swoimi przeciwieństwami, choć niezadeklarowane zmienne nieutworzone przez metody klas zachowują się nieco inaczej — mogą zostać przypisane i tym samym utworzone poza klasą pod `Private` (wszystkie niezadeklarowane zmienne są dostępne), jednak pod `Public` już nie (wszystkie niezadeklarowane zmienne są niedostępne).

Ponownie zachęcam do samodzielnego przestudiowania kodu w celu przekonania się, jak on działa. Warto zwrócić uwagę na to, że rozwiązanie to na górze dodaje dodatkowy, czwarty poziom zachowywania stanu, poza poziomami opisanymi w poprzednim podrozdziale. Funkcje sprawdzające wykorzystywane przez `lambda` są zapisywane w dodatkowym zakresie

zawierającym. Przykład ten napisany jest w taki sposób, by działał w Pythonie 2.x lub 3.x, choć ma pewne ograniczenie w wersji 3.x (wyjaśnione pokrótko w łańcuchu znaków dokumentacji pliku i nieco szerzej w opisie następującym po kodzie).

"""

Plik access2.py (wersje 3.x i 2.x)

Dekorator klasy z deklaracjami atrybutów jako prywatne oraz publiczne.

Kontroluje dostęp do atrybutów przechowywanych w instancji lub dziedziczonych przez nią po klasach. Private deklaruje nazwy atrybutów, których nie można pobrać lub przypisać z zewnątrz udekorowanej klasy. Public deklaruje nazwy atrybutów, które można pobrać lub przypisać z zewnątrz.

Uwaga: działa w Pythonie 3.x jedynie dla atrybutów o normalnych nazwach. Metody przeciążania operatorów __X__ wykonywane w sposób niejawny dla operacji wbudowanych nie wywołują metod __getattr__ ani __getattribute__ w klasach w nowym stylu. Należy tutaj dodać metody __X__ w celu przechwycenia i wydelegowania nazw wbudowanych.

"""

```
traceMe = False

def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)
            def __getattr__(self, attr):
                trace('pobranie:', attr)
                if failIf(attr):
                    raise TypeError('pobranie atrybutu prywatnego: ' + attr)
                else:
                    return getattr(self.__wrapped, attr)
            def __setattr__(self, attr, value):
                trace('ustawienie:', attr, value)
                if attr == '_onInstance__wrapped':
                    self.__dict__[attr] = value
                elif failIf(attr):
                    raise TypeError('modyfikacja atrybutu prywatnego: ' + attr)
                else:
                    setattr(self.__wrapped, attr, value)
        return onInstance
    return onDecorator
```

```

        return onInstance
    return onDecorator
def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))
def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))

```

Przykłady użycia można znaleźć w kodzie testu samosprawdzającego poprzedniego przykładu. Oto szybkie spojrzenie na działanie tych dekoratorów klas w sesji interaktywnej (działają one tak samo w Pythonie 2.x oraz 3.x z atrybutami o jawnych nazwach, jak testowane tutaj). Zgodnie z tym, co zapowiadaliśmy, nazwy nie prywatne lub publiczne można pobierać i modyfikować spoza podmiotowej klasy, natomiast nie można tego robić w przypadku nazw prywatnych lub niepublicznych.

```

>>> from access import Private, Public
>>> @Private('age')                                     # Person = Private('age')(Person)
... class Person:                                       # Person = onInstance ze stanem
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age                                # Próby dostępu z wewnętrz odbywają
się normalnie
...
>>> X = Person('Robert', 40)
>>> X.name                                         # Próby dostępu z zewnątrz są
sprawdzane
'Robert'
>>> X.name = 'Anna'
>>> X.name
'Anna'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tomasz'
TypeError: private attribute change: age
>>> @Public('name')
... class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...

```

```

>>> X = Person('robert', 40)                      # X jest instancją onInstance
>>> X.name                                     # onInstance osadza Person
'robert'
>>> X.name = 'Anna'
>>> X.name
'Anna'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tomasz'
TypeError: private attribute change: age

```

Szczegóły implementacji II

By pomóc w analizie kodu, poniżej znajduje się kilka finalnych uwag dotyczących tej wersji. Ponieważ jest ona jedynie uogólnieniem przykładu z poprzedniego podrozdziału, większość uwag, które go dotyczą, ma zastosowanie również tutaj.

Użycie nazw pseudoprzywatnych `_X`

Poza uogólnieniem wersja ta wykorzystuje również opcję nazw pseudoprzywatnych Pythona `_X` (z którą spotkaliśmy się w rozdziale 31.) do zlokalizowania atrybutu `wrapped` w klasie kontrolnej dzięki automatycznemu poprzedzeniu go nazwą klasy. Pozwala nam to uniknąć ryzyka konfliktów z atrybutem `wrapped`, który może być wykorzystywany przez prawdziwą, opakowaną klasę (co miało miejsce w poprzedniej wersji kodu) — może się to przydać w tak uniwersalnym narzędziu. Opcja ta nie zapewnia jednak prywatności, ponieważ taka nazwa może być swobodnie wykorzystywana poza klasą. Warto również zauważać, że w metodzie `__setattr__` będziemy musieli użyć łańcucha znaków pełnej, rozszerzonej nazwy (`'_onInstance_wrapped'`), ponieważ na to zmienia ją Python.

Złamanie prywatności

Choć powyższy przykład implementuje kontrolę dostępu dla atrybutów instancji oraz jej klas, można tę kontrolę na różne sposoby odwrócić — na przykład w sposób jawnym przechodząc rozszerzoną wersję atrybutu `wrapped` (`bob.pay` może nie zadziałać, ale pełna nazwa `bob._onInstance_wrapped.pay` już może!). Jeśli jednak musimy to robić w sposób jawnym, poziom kontroli powinien być wystarczający na potrzeby normalnego, zamierzzonego użycia. Oczywiście kontrolę prywatności można odwrócić w każdym języku programowania, o ile będziemy się wystarczająco mocno starać (w niektórych implementacjach C++ może także zadziałać `#define private public`). Choć kontrola dostępu może ograniczyć przypadkowe zmiany, w dowolnym języku programowania wiele zależy od programisty. W każdej sytuacji, w której kod źródłowy można zmienić, kontrola dostępu będzie mrzonką.

Kompromisy związane z dekoratorem

I znów, moglibyśmy uzyskać te same wyniki bez dekoratorów, wykorzystując funkcje zarządzające lub ręcznie pisząc kod ponownie dowiązujący nazwy dekoratorów. Składnia dekoratorów sprawia jednak, że całość jest nieco bardziej oczywista i spójna. Największą potencjalną wadą tego rozwiązania (i wszystkich innych opartych na obiektach opakowujących) jest to, że dostęp do atrybutów wymaga dodatkowego wywołania, a instancje udekorowanych klas nie są tak naprawdę instancjami oryginalnej dekorowanej klasy. Jeśli na przykład sprawdzimy ich typ za pomocą `X.__class__` czy `isinstance(X, C)`, okaże się, że są one

instancjami klasy *opakowującej*. O ile jednak nie planujemy wykonywać introspekcji na typach obiektów, kwestia ta jest najprawdopodobniej bez znaczenia, a dodatkowe wywołania będą miały miejsce głównie na etapie kodowania. Jak się przekonamy później, są sposoby automatycznego usuwania dekoratorów, gdy zajdzie taka potrzeba.

Znane problemy

W obecnej postaci przykład ten działa zgodnie z zamierzeniami w Pythonie 2.x oraz 3.x w przypadku metod jawnie wywoływanych za pomocą ich nazw. Jak to jednak ma miejsce w przypadku większości oprogramowania, zawsze coś można poprawić. Przede wszystkim to narzędzie obniża wydajność metod przeciążających operatory, jeżeli są one stosowane w klasach klienckich.

Zakodowana tu klasa pośrednicząca jest typową klasą w wersji 2.x, a w wersji 3.x jest klasą w nowym stylu. Dzięki temu kod obsługuje każdą klasę kliencką w wersji 2.x, ale w wersji 3.x nie weryfikuje ani nie deleguje metod przeciążających operatory i wywoływanych niejawnie podczas wykonywania wbudowanych operacji, chyba że zostaną przeddefiniowane w klasie pośredniczącej. W pełni obsługiwane są klienci, które nie wykorzystują przeciążonych operatorów, natomiast inne mogą wymagać w wersji 3.x napisania dodatkowego kodu.

Należy pamiętać, że nie jest to mankament klasy w nowym stylu, tylko wada *wersji* Pythona. Ten sam kod działa inaczej i do tego niepoprawnie tylko w wersji 3.x. Ponieważ natura klasy opakowywanego obiektu nie zależy od klasy pośredniczącej, interesuje nas tylko własny kod klasy pośredniczącej, który działa poprawnie w obu wersjach.

Na ten problem natknęliśmy się już kilka razy w tej książce, przeanalizujmy ogólnie jego wpływ na bardzo praktyczny kod, który tutaj napisaliśmy, oraz zbadajmy możliwości jego rozwiązania.

Ograniczenie: delegacja metod przeciążania operatorów kończy się niepowodzeniem w Pythonie 3.x

Tak jak wszystkie klasy oparte na delegacji, korzystające z metody `__getattribute__`, powyższy dekorator działa we wszystkich wersjach jedynie dla atrybutów o normalnych nazwach. Metody przeciążania operatorów, takie jak `__str__` oraz `__add__`, będą działały inaczej dla klas w nowym stylu i tym samym po wykonaniu kodu w Pythonie 3.x nie dotrą do osadzonego obiektu, jeśli są tam zdefiniowane.

Jak wiemy z poprzedniego rozdziału, klasy tradycyjne normalnie wyszukują nazwy przeciążania operatorów w instancjach w czasie wykonywania, jednak klasy w nowym stylu tego nie robią — całkowicie pomijają instancję i wyszukują metody tego typu w klasach. Tym samym metody przeciążania operatorów `X` wykonywane w sposób niejawny dla operacji wbudowanych *nie* wywołują ani metody `__getattr__`, ani `__getattribute__` dla klas w nowym stylu. Takie próby pobrania atrybutów pomijają całkowicie naszą metodę `onInstance.__getattr__`, dlatego nie da się ich sprawdzić ani wydelegować.

Nasza klasa dekoratora nie jest zapisana w kodzie jako klasa w nowym stylu (pochodząca od `object`), dlatego przechwyci metody przeciążania operatorów po wykonaniu w Pythonie 2.x. W wersji 3.x wszystkie klasy automatycznie są klasami w nowym stylu, więc metody tego typu *nie będą działały*, jeśli są one zapisane na osadzonym obiekcie, ponieważ nie będą przechwytywane przez klasę pośredniczącą.

Najprostsze obejście tego ograniczenia w Pythonie 3.x polega na ponownym, powtórzonym zdefiniowaniu w `onInstance` wszystkich metod przeciążania operatorów, które mogą być użyte w osadzonych obiektach. Takie dodatkowe metody można dodać ręcznie, za pomocą narzędzi po części automatyzujących to zadanie (na przykład dekoratorów klas lub metaklas omówionych w kolejnym rozdziale) lub za pomocą definicji w klasach nadzępnych. Choć jest to żmudne rozwiązanie, wymagające wpisania obszernego kodu, którego nie można tu zaprezentować, za chwilę poznamy sposoby spełnienia tego wymogu tylko w wersji 3.x.

By zobaczyć tę różnicę na własne oczy, można spróbować zastosować dekorator do klasy wykorzystującej metody przeciążania operatorów w Pythonie 2.x. Sprawdzanie działa jak poprzednio i zarówno metoda `__str__` wykorzystywana dla wyświetlania, jak i metoda `__add__` wykonywana dla operatora + wywołują metodę `__getattr__` dekoratora i tym samym zostaną poprawnie sprawdzone i wydelegowane do podmiotowego obiektu klasy Person.

```
C:\code> c:\python27\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
...         self.age = 42
...     def __str__(self):
...         return 'Osoba: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()
>>> X.age                                     # Sprawdzanie poprawności nie
                                                powiedzie się (zgodnie z planem)
TypeError: pobranie atrybutu prywatnego: age
>>> print(X)                                 # __getattr__ => wykonuje
Person.__str__
Osoba: 42
>>> X + 10                                    # __getattr__ => wykonuje
Person.__add__
>>> print(X)                                 # __getattr__ => wykonuje
Person.__str__
Osoba: 52
```

Gdy jednak ten sam kod wykonamy w Pythonie 3.x, wywoływane w niejawnny sposób metody `__str__` oraz `__add__` pomijają metodę `__getattr__` dekoratora i szukają definicji w samej klasie dekoratora oraz ponad nią. Instrukcja `print` odnajduje domyślny sposób wyświetlania odziedziczony po typie klasy (a tak naprawdę po domniemanej klasie nadrzędnej `object` w Pythonie 3.x), natomiast operator + generuje błąd, ponieważ nie odziedziczył żadnej wartości domyślnej.

```
C:\code> c:\python33\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
...         self.age = 42
```

```

...     def __str__(self):
...         return 'Osoba: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()                      # Sprawdzanie poprawności nazw nadal
działa
>>> X.age                            # Jednak Python 3.x nie deleguje nazw
wbudowanych!
TypeError: pobranie atrybutu prywatnego: age
>>> print(X)
<access2.accessControl.<locals>.onDecorator.<locals>.onInstance object at
...etc>
>>> X + 10
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
>>> print(X)
<access2.accessControl.<locals>.onDecorator.<locals>.onInstance object at
...etc>

```

O dziwo, tak się dzieje tylko podczas kierowania wbudowanych operacji. Jawne, bezpośrednie wywołania przeciążonych metod są kierowane do metody `__getattr__`. Nie należy jednak oczekwać, że klienci wykorzystujące przeciążanie operatora będą robić to samo:

```

>>> X.__add__(10)                      # Wywołanie za pomocą nazwy działa
normalnie.
>>> X._onInstance_wrapped.age        # Naruszenie prywatności w celu
wyświetlenia wyników.

```

52

Innymi słowy, wynika to z różnic pomiędzy *wbudowanymi operacjami a jawnie wywoływanymi metodami*. Rzeczywiste nazwy metod mają tu niewielkie znaczenie. Tylko w klasach w nowym stylu w wersji 3.x pomijany jest jeden krok.

Użycie alternatywnej metody `__getattribute__` nic tu nie pomoże. Choć jest ona zdefiniowana w taki sposób, by przechwytywać każdą referencję do atrybutu (a nie tylko dla nazw niezdefiniowanych), nie jest wykonywana przez operacje wbudowane. Opcja `property` Pythona (omówiona w rozdziale 38.) także nam nie pomoże. Warto przypomnieć, że właściwości to kod wykonywany automatycznie, powiązany z określonymi atrybutami, definiowany w czasie pisania klasy. Nie są one zaprojektowane do obsługi dowolnych atrybutów w opakowanych obiektach.

Sposoby redefiniowania metod przeciążających operatory w wersji 3.x

Jak wspomniano wcześniej, najprostszym rozwiązaniem w Pythonie 3.x jest powtórne zdefiniowanie nazw przeciążających operatory, które mogą się pojawić w obiektach osadzonych w klasach opartych na delegacji, takich jak nasz dekorator. Nie jest to rozwiązanie idealne, ponieważ powoduje pewną powtarzalność kodu, zwłaszcza w porównaniu z rozwiązaniem przeznaczonym dla Pythona 2.x. Nie jest to jednak również zbyt duży wysiłek programistyczny, można go do pewnego stopnia zautomatyzować za pomocą narzędzi lub klas nadrzędnych, co

wystarczy do umożliwienia działania dekoratora w Pythonie 3.x i pozwoli na deklarowanie jako prywatne lub publiczne również nazw przeciążających operatory (przy założeniu, że każda metoda przeciążania operatora wewnętrznie wykonuje test `failIf`).

Definicja śródwierszowa

Poniżej przedstawione jest rozwiążanie wykorzystujące definicję śródwierszową. W klasie pośredniczącej ponownie zdefiniowane są wszystkie metody przeciążające operatory, które opakowany obiekt może samodzielnie definiować, przechwytywać i delegować. Dla celów ilustracyjnych zostały dodane cztery metody przechwytyjące operatory. W przypadku innych operatorów należy postąpić w taki sam sposób (nowy kod jest wyróżniony pogrubioną czcionką).

```
def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)
                # Przechwycenie i wydelegowanie metod przeciążających operatory
            def __str__(self):
                return str(self.__wrapped)
            def __add__(self, other):
                return self.__wrapped + other          # Lub getattr(x,
'__add__')(y)
            def __getitem__(self, index):
                return self.__wrapped[index]          # W miarę potrzeb
            def __call__(self, *args, **kargs):
                return self.__wrapped(*arg, *kargs)    # W miarę potrzeb
            #... i ewentualne inne potrzebne metody...
            def __getattr__(self, attr):
                ...
            def __setattr__(self, attr, value):
                ...
        return onInstance
    return onDecorator
```

Nadrzędne klasy mieszane

Alternatywnym rozwiążaniem jest wstawienie powyższych metod za pomocą wspólnej *klasy nadrzędnnej*. Jeżeli takich metod jest kilkanaście, wtedy lepiej do tego zadania nadaje się zewnętrzna klasa, szczególnie jeżeli jest na tyle uniwersalna, że można ją wykorzystywać w dowolnym interfejsie klasy pośredniczącej. Do przechwytywania i delegowania wbudowanych

operacji wystarczy zastosować jedną z poniższych (spośród wielu innych możliwych) nadrzędnych klas mieszanych:

- Pierwsza klasa przechytuje wbudowane operacje i kieruje je do metody `__getattribute__` podklasy. Wymagane jest przy tym, aby nazwy przeciążające operatory były publiczne, zgodnie ze specyfikacją dekoratora. Wbudowane operacje będą jednak obsługiwane tak samo, niezależnie od tego, czy będą wywoływane jawnie za pomocą nazwy, czy typowej klasy w wersji 2.x.
- Druga klasa przechytuje wbudowane operacje i kieruje je bezpośrednio do opakowanego obiektu. Wymagany jest dostęp do atrybutu pośredniczącego `_wrapped` przy założeniu, że można za jego pomocą dostać się do osadzonego obiektu. Jest to rozwiązanie dalekie od ideału, ponieważ uniemożliwia opakowanym obiektom używanie tej samej nazwy, jak również tworzy zależność od klasy nadzędnej. Jest jednak lepsze od korzystania z zagmatwanego atrybutu `_onInstance_wrapped` i nie gorsze od metody o podobnej nazwie.

Podobnie jak w przypadku definicji śródwierniowej, każda z klas mieszanych wymaga utworzenia w ogólnych narzędziach pełniących role pośredników dla interfejsów dowolnych obiektów po jednej metodzie na operację wbudowaną. Zwróćmy uwagę, jak klasy te przechytują *wywołania* operacji, a nie *pobrania* atrybutów i z tego względu muszą wykonywać właściwe operacje poprzez delegowanie wywołania lub wyrażenia:

```
class BuiltinsMixin:  
    def __add__(self, other):  
        return self.__class__.__getattribute__(self, '__add__')(other)  
    def __str__(self):  
        return self.__class__.__getattribute__(self, '__str__')()  
    def __getitem__(self, index):  
        return self.__class__.__getattribute__(self, '__getitem__')(index)  
    def __call__(self, *args, **kargs):  
        return self.__class__.__getattribute__(self, '__call__')(*args, **kargs)  
        # ...i wiele innych potrzebnych operacji  
    def accessControl(failIf):  
        def onDecorator(aClass):  
            class onInstance(BuiltinsMixin):  
                ...pozostały kod bez zmian...  
                def __getattribute__(self, attr): ...  
                def __setattr__(self, attr, value): ...  
        class BuiltinsMixin:  
            def __add__(self, other):  
                return self._wrapped + other                                # Założenie, że jest  
atrybut _wrapped  
            def __str__(self):  
                __getattribute__
```

```

        return str(self._wrapped)

    def __getitem__(self, index):
        return self._wrapped[index]

    def __call__(self, *args, **kargs):
        return self._wrapped(*args, **kargs)
        # ...i wiele innych potrzebnych operacji

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            ...Należy użyć self._wrapped zamiast self.__wrapped...
            def __getattr__(self, attr): ...
            def __setattr__(self, attr, value): ...

```

Każda z powyższych klas mieszanych będzie zewnętrznym kodem, ale musi być zaimplementowana tylko raz. Wydaje się to znacznie prostszym rozwiązaniem niż różne dostępne w internecie sposoby wykorzystujące *metaklasy* lub narzędzia oparte na dekoratorach, niepotrzebnie wypełniające każdą klasę pośredniczącą wymaganymi metodami. (Zasady działania takich narzędzi opisane są w przykładach modyfikacji klasy w rozdziale 40.).

Warianty kodowania: routery, deskryptory, automatyzacja

Oczywiście obie opisane w poprzedniej sekcji nadrzędne klasy mieszane można udoskonalić, wprowadzając dodatkowe, obszernie tu opisane zmiany w kodzie. Wyjątkiem są dwa warianty, o których warto krótko wspomnieć. Najpierw porównajmy poniższą odmianę *pierwszej* klasy mieszanej, w której wykorzystany jest kod o prostszej strukturze, ale wykonujący dodatkowe operacje przy każdym wywołaniu, przez co działa wolniej (choć nieznacznie w kontekście pośrednika):

```

class BuiltinsMixin:

    def reroute(self, attr, *args, **kargs):
        return self.__class__.__getattr__(self, attr)(*args, **kargs)

    def __add__(self, other):
        return self.reroute('__add__', other)

    def __str__(self):
        return self.reroute('__str__')

    def __getitem__(self, index):
        return self.reroute('__getitem__', index)

    def __call__(self, *args, **kargs):
        return self.reroute('__call__', *args, **kargs)
        # ...i wiele innych potrzebnych operacji

```

Teraz w obu powyższych klasach mieszanych metody przeciążające operatory są zakodowane *jawnie* i przechwytyują *wywołania* operacji. Stosując alternatywne kodowanie, moglibyśmy mechanicznie *generować* metody na podstawie listy nazw i *przechwytywać* tylko pobieranie

atributów przed wywołaniem, tworząc *deskryptory* na poziomie klasy, jak w poprzednim rozdziale. Ilustruje to poniższy kod, w którym — podobnie jak w drugiej alternatywie klasy mieszanej — przyjęte jest założenie, że obiekt pośredniczący ma nazwę `_wrapped` w samej instancji pośredniczącej:

```
class BuiltinsMixin:

    class ProxyDesc(object):          # Obiekty dla wersji 2.x

        def __init__(self, attrname):
            self.attrname = attrname

        def __get__(self, instance, owner):
            return getattr(instance._wrapped, self.attrname) # Założenie, że
            jest atrybut _wrapped

        builtins = ['add', 'str', 'getitem', 'call']           # Plus dowolne
        inne operacje

        for attr in builtins:
            exec('__%s__ = ProxyDesc("__%s__")' % (attr, attr))
```

Powyższy kod jest prawdopodobnie najbardziej zwięzły, ale też najbardziej niejawny i skomplikowany, jak również silnie powiązany ze swoimi podklasami za pomocą wspólnej nazwy. Pętla na końcu klasy jest odpowiednikiem poniższego kodu. Działa w lokalnym zakresie klasy mieszanej i tworzy deskryptory odpowiadające na pierwsze wyszukiwania nazw poprzez pobieranie z nowego obiektu opakowanego w `__get__`, a nie samodzielne przechwytywanie wykonywanych później operacji:

```
__add__ = ProxyDesc("__add__")
__str__ = ProxyDesc("__str__")
...itd....
```

Po dodaniu takich metod przeciążających operatory — czy to za pomocą definicji śródwierszowych, czy dziedziczenia klas mieszanych — prywatny klient z poprzedniego przykładu, przeciążający operator dodawania oraz instrukcję `print` za pomocą metod `__str__` i `__add__`, działa poprawnie w wersjach 2.x i 3.x, podobnie jak podklasy przeciążające indeksy i wywołania. Do dalszych eksperymentów można wykorzystać dołączone do książki pliki `access2_builtin*.py` zawierające pełne kody powyższych opcji. Trzecia opcja z klasami mieszonymi jest również wykorzystana w rozwiązaniu quizu na końcu rozdziału.

Czy metody operatorów należy weryfikować?

Aby poprawnie delegować wywołania, niezbędne jest dodanie obsługi metod przeciążających operatory. Jednak w naszym konkretnym przypadku prywatności pojawia się kilka opcji projektowych. W szczególności prywatność metod przeciążających operatory jest różna w zależności od implementacji:

- Ponieważ wywołują one metodę `__getattr__`, klasa mieszana wymaga, aby wszystkie udostępniane nazwy `_X_` były wymienione w dekoracjach `Public`, albo zamiast tego były użyte dekoracje prywatne, jeżeli przeciążanie operatorów odbywa się u klientów. W klasach intensywnie wykorzystujących przeciążanie dekorator `Public` jest niepraktyczny.
- Ponieważ całkowicie omijają one metodę `__getattr__` jak w powyższym kodzie, to zarówno definicje śródwierszowe, jak i klasy mieszane `self._wrapped` nie mają powyższych ograniczeń, ale uniemożliwiają definiowanie wbudowanych operacji jako prywatnych, przez co operacje te są kierowane asymetrycznie, zarówno jawnie wywoływane za pomocą nazw `_X_`, jak i zwykłych klas w wersji 2.x.

- Zwykłe klasy w wersji 2.x mają ograniczenia wymienione w pierwszym punkcie, ponieważ nazwy `__X__` są kierowane automatycznie przez metodę `__getattr__`.
- Nazwy i protokoły nazw przeciążających są różne w wersjach 2.x i 3.x, przez co dekorowanie niezależne od wersji nie jest trywialnym zadaniem (np. dekoratory `Public` mogą wymagać podania list nazw w obu wersjach).

W tym miejscu ostateczną politykę pozostawiamy do zdefiniowania, jednak niektóre interfejsy mogą oczekwać, aby delegowane nazwy operatorów `__X__` były zawsze przekazywane bez sprawdzania.

Jednak w ogólnym przypadku niezbędne jest wpisanie dodatkowego kodu, aby klasy w nowym stylu w wersji 3.x można było wykorzystywać jako pośredników delegacji. Z zasady *każda* metoda przeciążająca operator, która nie jest już automatycznie kierowana jako zwykły atrybut instancji, musi być dodatkowo zdefiniowana w klasie ogólnych narzędzi, tak jak dekorator prywatności. Z tego powodu to rozszerzenie zostało pominięte w naszym kodzie — potencjalnie jest ponad 50 takich metod! Ponieważ w wersji 3.x wszystkie klasy są w nowym stylu, kodowanie oparte na delegowaniu jest bardziej trudne, niemniej jednak możliwe.

Alternatywy implementacyjne: wstawianie `__getattribute__`, inspekcja stosu wywołań

Choć powtarzanie definicji metod przeciążania operatorów w obiektach opakowujących jest najprawdopodobniej najprostszym obejściem problemu z Pythonem 3.x zarysowanego w tym podrozdziale, niekoniecznie musi to być jedyne rozwiązanie. Nie mamy tutaj miejsca na dalsze omawianie tej kwestii, dlatego zbadanie innych potencjalnych rozwiązań pozostawiam jako sugerowane ćwiczenie. Ponieważ jednak jedno z rozwiązań dobrze opisuje ogólne koncepcje związane z klasami, zasługuje na krótkie omówienie.

Jedną z wad tego przykładu jest to, że obiekty instancji nie są tak naprawdęinstancjami oryginalnej klasy — są zamiast tego instancjami klasy *opakowującej*. W niektórych programach polegających na sprawdzaniu typów może to mieć znaczenie. By obsługiwać tego typu przypadki, możemy spróbować osiągnąć ten sam efekt, *wstawiając* metodę `__getattribute__` do oryginalnej klasy w celu przechwycenia *wszystkich* referencji do atrybutów wykonywanych na instancjach. Wstawiona metoda przekazałaby poprawne żądania w górę do klasy nadzędnej w celu uniknięcia pętli za pomocą technik omówionych w poprzednim rozdziale. Oto potencjalna zmiana, jaką należy wprowadzić do kodu naszego dekoratora klasy.

```
# Wstawienie metody. Pozostała część kodu access2.py jest taka sama jak
# poprzednio

def accessControl(failIf):
    def onDecorator(aClass):
        def getattributes(self, attr):
            trace('pobranie:', attr)
            if failIf(attr):
                raise TypeError('pobranie atrybutu prywatnego: ' + attr)
            else:
                return object.__getattribute__(self, attr)
        def setattributes(self, attr, value):
            trace('set:', attr)
            if failIf(attr):
                pass
            else:
                self.__dict__[attr] = value
        aClass.__getattribute__ = getattributes
        aClass.__setattribute__ = setattributes
        return aClass
    return onDecorator
```

```

        raise TypeError('private attribute change: ' + attr)
    else:
        return object.__setattr__(self, attr, value)
aClass.__getattribute__ = getattributes
aClass.__setattr__ = setattr # Wstawienie metod
dostępowych
return aClass # Zwrócenie oryginalnej
klasy
return onDecorator

```

Powyższa alternatywa rozwiązuje problem sprawdzania typów, jednak ma inne ograniczenia. Przede wszystkim takiego dekoratora można używać tylko z klientami *klas w nowym stylu*. Ponieważ metoda `__getattribute__` (podobnie jak `__setattr__`) jest narzędziem stosowanym tylko w tego rodzaju klasach, w wersji 2.x dekorowane klasy muszą wykorzystywać dziedziczenie w nowym stylu, co może — ale nie musi — być zgodne z ich przeznaczeniem. W rzeczywistości zbiór obsługiwanych klas jest jeszcze bardziej ograniczony. Wstawianie metod zakłoca działanie klientów, które samodzielnie korzystają z metod `__setattr__` lub `__getattribute__`.

Dodatkowo rozwiązanie to nie rozwiązuje problemu atrybutów operacji *wbudowanych*, omówionego w poprzednim podrozdziale, ponieważ metoda `__getattribute__` nie działa w tych kontekstach. W naszym przypadku, gdyby klasa Person miała metodę `__str__`, zostałaby ona wykonana przez operacje wyświetlania, jednak tylko dlatego, że byłaby obecna w tej klasie. Tak jak wcześniej, atrybut `__str__` nie zostałby przekierowany do wstawionej metody `__getattribute__` w sposób uniwersalny — wyświetlanie całkowicie obeszłoby tę metodę i powodowałoby bezpośrednie wywołanie metody `__str__` klasy.

Choć i tak jest to najprawdopodobniej lepsze od całkowitego braku obsługi metod przeciążających operatorów w opakowanym obiekcie (pomijając ich redefinicję), rozwiązanie to nadal nie jest w stanie przechwytywać i sprawdzać metod `__X__`, przez co żadna z nich nie może być prywatna. Choć większość metod przeciążania operatorów z założenia ma być publiczna, niektóre wcale nie muszą takie być.

Co gorsza, ponieważ rozwiązanie bez opakowywania działa dzięki dodaniu metod `__getattribute__` i `__setattr__` do udekorowanej klasy, przechwytuje również próby dostępu do atrybutów wykonywane przez samą klasę i sprawdza je w ten sam sposób jak próby dostępu wykonywane z zewnątrz — co oznacza, że metody klasy również nie będą w stanie wykorzystywać zmiennych prywatnych! Jest to przeszkoda dyskwalifikująca rozwiązanie polegające na wstawianiu metod.

Tak naprawdę wstawienie metod w ten sposób jest funkcjonalnym odpowiednikiem ich *dziedziczenia* i wiąże się z tymi samymi ograniczeniami co kod tworzący prywatność atrybutów z rozdziału 30. By dowiedzieć się, czy próba dostępu odbywa się z wewnątrz klasy, czy z zewnątrz, nasza metoda może być zmuszona sprawdzać obiekty ramek lub *stos wywołań* Pythona. Może to prowadzić do pewnego rozwiązania problemu (na przykład zastąpienia atrybutów prywatnych właściwościami lub deskryptorami sprawdzającymi *stos*), ale w dalszym ciągu spowalniałyby dostęp i jest zbyt zagmatwane, byśmy to tutaj omawiali. (Wygląda na to, że deskryptory potrafią robić wszystko, nawet to, czego nie powinny!).

Choć technika wstawiania metod jest interesująca i może mieć znaczenie w innych przypadkach użycia, nie sprostała jednak naszym celom. Nie będziemy dalej zajmować się tym wzorcem kodu, ponieważ w kolejnym rozdziale techniki rozszerzania klas omówimy w połączeniu z metaklasami. Jak zobaczymy, metaklasy nie są ściśle wymagane do tego typu modyfikacji klas, ponieważ tę samą rolę mogą często pełnić dekoratory klas.

W Pythonie nie chodzi o kontrolę

Skoro już podjąłem tak wielki wysiłek w celu dodania deklaracji atrybutów jako prywatnych i publicznych do kodu napisanego w Pythonie, muszę raz jeszcze przypomnieć, że dodawanie tego typu kontroli dostępu do klas nie jest w pełni *pythonowym* sposobem działania. Tak naprawdę większość programistów Pythona uzna najprawdopodobniej ten przykład za w dużej mierze (lub całkowicie) zbędny — poza pełnieniem roli demonstracji działania dekoratorów w praktyce. Większość dużych programów w Pythonie świetnie sobie radzi bez żadnej tego typu kontroli.

Oznacza to, że możliwości wykorzystania takiego narzędzia w programowaniu są dość ograniczone. Jeśli jednak chcemy regulować dostęp w celu wyeliminowania błędów programistycznych lub jesteśmy prawie byłym programistą C++ lub Javy, większość zadań można zrealizować za pomocą przeciążania operatorów i narzędzi introspekcji Pythona.

Przykład — sprawdzanie poprawności argumentów funkcji

W ostatnim przykładzie użyteczności dekoratorów w niniejszym podrozdziale napiszemy *dekorator funkcji*, który automatycznie sprawdza, czy argumenty przekazane do funkcji bądź metody mieścią się w określonym przedziale liczbowym. Został on zaprojektowany w celu użycia na etapie programowania lub produkcji i można go wykorzystać jako szablon dla innych, podobnych zadań (na przykład sprawdzania typów argumentów, jeśli musimy je wykonywać). Ponieważ przekroczyliśmy już limit objętości dla tego rozdziału, kod z tego przykładu będzie w dużej mierze materiałem do samodzielnego przestudiowania, z ograniczoną częścią opisową. Jak zwykle więcej szczegółów można znaleźć, przeglądając sam kod.

Cel

W omówieniu programowania zorientowanego obiektowo w rozdziale 28. napisaliśmy klasę, która przyznawała podwyżkę obiektom reprezentującym ludzi w oparciu o przekazaną wartość procentową.

```
class Person:  
    ...  
    def giveRaise(self, percent):  
        self.pay = int(self.pay * (1 + percent))
```

Zauważmy tam, że gdybyśmy chcieli, by kod był bardziej rozbudowany, nieźleym pomysłem byłoby sprawdzenie, czy wartość procentowa nie jest zbyt duża lub zbyt mała. Moglibyśmy wprowadzić tego typu sprawdzanie za pomocą instrukcji `if` lub `assert wstawionych` do samej metody.

```
class Person:  
    def giveRaise(self, percent): # Sprawdzenie w kodzie metody  
        if percent < 0.0 or percent > 1.0:  
            raise TypeError, 'niepoprawna wartość procentowa'  
        self.pay = int(self.pay * (1 + percent))
```

```

class Person:                                     # Sprawdzenie za pomocą
    instrukcji assert

    def giveRaise(self, percent):
        assert percent >= 0.0 and percent <= 1.0, 'niepoprawna wartość
procentowa'

        self.pay = int(self.pay * (1 + percent))

```

Takie rozwiązanie zanieczyszcza jednak metodę testami wewnętrznyimi, które najprawdopodobniej przydadzą się jedynie w trakcie pisania programu. W bardziej skomplikowanych przypadkach może się to stać dość żmudne (wyobraźmy sobie próby wstawiania kodu potrzebnego do zaimplementowania prywatności atrybutów z dekoratora z poprzedniego podrozdziału). Co jednak gorsze, jeśli logika sprawdzająca będzie się musiała kiedykolwiek zmienić, potencjalnie będziemy musieli odnaleźć i aktualnić dowolnie dużą liczbę wierszy.

Ciekawszą i bardziej przydatną alternatywą byłoby napisanie narzędzia ogólnego przeznaczenia, które jest w stanie automatycznie wykonywać dla nas testy przedziałów dla argumentów dowolnej funkcji bądź metody, jaką utworzymy teraz lub w przyszłości. Rozwiązanie z *dekoratorem* sprawia, że będzie się to odbywało w sposób spójny i jawnym.

```

class Person:

    @rangetest(percent=(0.0, 1.0))                      # Użycie dekoratora do
    sprawdzania

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

```

Wyizolowanie logiki sprawdzającej poprawność w dekoratorze upraszcza zarówno kod klienta, jak i późniejsze utrzymywanie.

Warto zauważyc, że nasz cel jest tutaj inny od sytuacji ze sprawdzaniem poprawności atrybutów z ostatniego przykładu z poprzedniego rozdziału. Tutaj chcemy sprawdzić wartości *argumentów funkcji* przy przekazywaniu, a nie wartości atrybutów przy ustawianiu. Dekorator Pythona wraz z narzędziami do introspekcji pozwolą nam wdrożyć to nowe zadanie równie łatwo.

Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych

Zacznijmy od prostej implementacji testu przedziału. Dla uproszczenia zaczniemy od napisania kodu dekoratora, który działa jedynie dla argumentów pozycyjnych i zakłada, że zawsze, w każdym wywołaniu pojawiają się one na tej samej pozycji. Nie mogą być przekazane po nazwie słowa kluczowego i nie obsługujemy dodatkowych słów kluczowych ***args* w wywołaniach, ponieważ może to zmienić pozycje zadeklarowane w dekoratorze. Zapiszmy poniższy kod w pliku o nazwie *rangetest1.py*:

```

def rangetest(*argchecks):                      # Sprawdzenie przedziałów argumentów
    pozycyjnych

    def onDecorator(func):                         # True, jeśli "python -O main.py
                                                # Nie działa: bezpośrednie wywołanie
                                                # oryginału
        if not __debug__:
            args...""
        return func

```

```

        else:                                # Inaczej opakowanie w czasie
debugowania

    def onCall(*args):
        for (ix, low, high) in argchecks:
            if args[ix] < low or args[ix] > high:
                errmsg = 'Argument %s nie mieści się w przedziale %s..%s' %
(ix, low, high)
                raise TypeError(errmsg)
        return func(*args)
    return onCall
return onDecorator

```

W obecnej postaci powyższy kod jest właściwie powtóżeniem omówionych wcześniej wzorców kodu. Wykorzystujemy między innymi argumenty dekoratora i zagnieżdżone zakresy na potrzeby zachowywania stanu.

Używamy również zagnieżdżonych instrukcji `def`, by przykład działał zarówno dla prostych funkcji, jak i dla metod, zgodnie z tym, czego nauczyliśmy się wcześniej. Po wykorzystaniu dla metody klasy `onCall` otrzymujemy instancję klasy podmiotowej w pierwszym elemencie `*args` i przekazuje ją dalej do `self` w oryginalnej funkcji metody. Liczby dla argumentów w testach przedziałów zaczynają się w tym przypadku od 1, a nie od 0.

Warto również zauważyć, że kod ten wykorzystuje wbudowaną zmienną `__debug__`. Python ustawia ją na `True`, o ile nie wykonujemy kodu z ustawioną opcją wiersza poleceń `-O` oznaczającą optymalizację (na przykład `python -O main.py`). Kiedy zmienna `__debug__` zwraca `False`, dekorator zwraca oryginalną funkcję bez zmian w celu uniknięcia dodatkowych wywołań i związanego z tym negatywnego wpływu na wydajność. Innymi słowy, w przypadku użycia argumentu `-O` dekorator automatycznie *usuwa* modyfikujący kod, dzięki czemu nie trzeba tego robić ręcznie.

Pierwsza wersja naszego rozwiązania będzie wykorzystywana w następujący sposób:

```

# Plik rangetest1_test.py

from __future__ import print_function      # Wersja 2.x

from rangetest1 import rangetest

print(__debug__)                          # False, jeśli "python -O main.py"
@rangetest((1, 0, 120))                  # persinfo = rangetest(...)

def persinfo(name, age):                 # age musi się mieścić w przedziale
0..120

    print('%s ma %s lat' % (name, age))

@rangetest([0, 1, 31], [1, 1, 12], [2, 0, 2009])

def birthday(D, M, Y):
    print('Data urodzenia = {0}/{1}/{2}'.format(D, M, Y))

class Person:

    def __init__(self, name, job, pay):

```

```

        self.job = job
        self.pay = pay
    @rangetest([1, 0.0, 1.0])           # giveRaise = rangetest(...)
(giveRaise)
    def giveRaise(self, percent):      # Argument 0 to tutaj instancja self
        self.pay = int(self.pay * (1 + percent))

# Wiersze z komentarzem zgłasząją błąd TypeError, o ile w wierszu polecen
powłoki nie wykorzystano "python -0"
persinfo('Robert Zielony', 45)       # Tak naprawdę wykonuje onCall(...)
ze stanem
#persinfo('Robert Zielony', 200)      # Lub person, jeśli
ustawiono argument wiersza polecen -0
birthday(31, 5, 1963)
#birthday(32, 5, 1963)
anna = Person('Anna Czerwona', 'programista', 100000)
anna.giveRaise(.10)                  # Tak naprawdę wykonuje
onCall(self, .10)
print(anna.pay)                    # Lub giveRaise(self, .10), jeśli z
-0
#anna.giveRaise(1.10)
#print(anna.pay)

```

Po wykonaniu poprawne wywołania w powyższym kodzie zwracają następujące wyniki. Cały kod z tego podrozdziału działa tak samo w Pythonie 2.x oraz 3.x, ponieważ dekoratory funkcji obsługiwane są w obu wersjach, nie korzystamy z delegacji atrybutów, za to używamy niezależnych od wersji konstrukcji wyjątków i wyświetlania treści.

```
C:\code> python rangetest1_test.py
True
Robert Zielony ma 45 lat
Data urodzenia = 31/5/1963
110000
```

Usunięcie komentarzy z któregoś z niepoprawnych wywołań powoduje zgłoszenie błędu `TypeError` przez dekorator. Oto wynik po pozwoleniu na wykonanie dwóch ostatnich wierszy (jak zwykle pominąłem część tekstu komunikatu o błędzie w celu zaoszczędzenia miejsca).

```
C:\code> python rangetest1_test.py
True
Robert Zielony ma 45 lat
Data urodzenia = 31/5/1963
110000
TypeError: Argument 1 nie mieści się w przedziale 0.0...1.0
```

Uruchomienie Pythona z opcją wiersza poleceń `-O` wyłączy sprawdzanie przedziałów, jednak pozwoli także uniknąć pogorszenia wydajności związanego z warstwą opakowującą. W rezultacie wywołujemy oryginalną, nieudekorowaną funkcję w sposób bezpośredni. Zakładając, że narzędzie to służy nam tylko do debugowania, możemy wykorzystać tę opcję do zoptymalizowania programu przed użyciem go w środowisku produkcyjnym.

```
C:\code> python -O rangetest1_test.py
```

```
False
```

```
Robert Zielony ma 45 lat
```

```
Data urodzenia = 31/5/1963
```

```
110000
```

```
231000
```

Uogólnienie kodu pod kątem słów kluczowych i wartości domyślnych

Poprzednia wersja kodu ilustruje podstawy, z których musimy skorzystać, jednak jest stosunkowo ograniczona. Obsługuje jedynie sprawdzanie poprawności argumentów przekazanych po pozycji i nie sprawdza argumentów ze słowami kluczowymi. Tak naprawdę zakłada, że żadne słowa kluczowe nie będą przekazywane w sposób zakłócający pozycję argumentów. Dodatkowo nie robi nic z argumentami z wartościami domyślnymi, które można pominąć w wywołaniu. Takie rozwiązanie będzie w porządku, jeśli wszystkie argumenty przekazywane są po pozycji i nigdy nie mają wartości domyślnych, ale jest to dalekie od ideału w przypadku narzędzia ogólnego przeznaczenia. Python obsługuje o wiele bardziej elastyczne tryby przekazywania argumentów, które nasz kod na razie pomija.

Odmiana naszego przykładu zaprezentowana poniżej radzi sobie nieco lepiej. Dopasowując oczekiwane argumenty opakowanej funkcji do prawdziwych argumentów przekazanych w wywołaniu, pozwala obsługiwać sprawdzanie poprawności przedziałów dla argumentów przekazywanych albo po pozycji, albo za pomocą słowa kluczowego. Pomijane jest testowanie argumentów domyślnych pominiętych w wywołaniu. Mówiąc w skrócie, argumenty do sprawdzenia są określone przez słowa kluczowe dla dekoratora, który później przechodzi zarówno krótką pozycyjną `*args`, jak i słownik słów kluczowych `**kargs` w celu sprawdzenia poprawności.

```
"""
```

```
Plik rangetest.py
```

Dekorator funkcji wykonujący sprawdzanie poprawności przedziałów dla przekazanych argumentów. Argumenty dla dekoratora określone są po słowach kluczowych. W samym wywołaniu argumenty mogą być przekazywane po pozycji lub za pomocą słowa kluczowego. Wartości domyślne mogą być pomijane.

Przykładowe przypadki użycia znajdują się w pliku `rangetest_test.py`.

```
"""
```

```
trace = True
```

```
def rangetest(**argchecks):      # Sprawdzenie poprawności przedziałów dla
    obu oraz wartości domyślnych
```

```
    def onDecorator(func):        # onCall pamięta func oraz argchecks
```

```

        if not __debug__:           # True, jeśli ustawiono "python -O main.py
args..."                        

        return func               # Opakowanie przy debugowaniu; inaczej
użycie oryginału

    else:

        code = func.__code__
        allargs = code.co_varnames[:code.co_argcount]
        funcname = func.__name__
        def onCall(*pargs, **kargs):
            # Wszystkie argumenty pozycyjne pargs dopasowują pierwsze N
oczekiwanych argumentów po pozycji

            # Reszta musi być w kargs lub jest pomijanymi wartościami
domyślnymi

            expected = list(allargs)
            positionals = positionals[:len(pargs)]
            for (argname, (low, high)) in argchecks.items():
                # Dla wszystkich argumentów, które mają być sprawdzone
                if argname in kargs:
                    # Przekazane po nazwie
                    if kargs[argname] < low or kargs[argname] > high:
                        errmsg = '{0} argument "{1}" nie mieści się w przedziale
{2}..{3}'
                        errmsg = errmsg.format(funcname, argname, low, high)
                        raise TypeError(errmsg)

                elif argname in positionals:
                    # Przekazane po pozycji
                    position = positionals.index(argname)
                    if pargs[position] < low or pargs[position] > high:
                        errmsg = '{0} argument "{1}" nie mieści się w przedziale
{2}..{3}'
                        errmsg = errmsg.format(funcname, argname, low, high)
                        raise TypeError(errmsg)

                else:
                    # Zakładamy, że nie został przekazany: wartość domyślna
                    if trace:
                        print('Argument "{0}" ma wartość
domyślną'.format(argname))

```

```

        return func(*pargs, **kargs)           # OK: wykonanie
oryginalnego wywołania

    return onCall

    return onDecorator

```

Poniższy skrypt testowy pokazuje, w jaki sposób wykorzystywany jest dekorator. Argumenty do sprawdzenia przekazywane są jako argumenty dekoratora ze słowami kluczowymi, natomiast w samym wywołaniu możemy przekazać je albo po nazwie, albo po pozycji i pominąć argumenty z wartościami domyślnymi, nawet jeśli mają one być sprawdzane w inny sposób.

```
"""
```

Plik rangetest_test.py (wersje 3.x i 2.x)

Wiersze z komentarzami zgłaszą błąd TypeError, o ile nie ustawiono "python -O" w wierszu poleceń powłoki.

```
"""
```

```

from __future__ import print_function # Wersja 2.x
from devtools import rangetest

# Testowanie funkcji, pozycyjne i po słowach kluczowych
@rangetest(age=(0, 120))           # persinfo = rangetest(...)
(persinfo)

def persinfo(name, age):
    print('%s ma %s lat' % (name, age))

@rangetest(D=(1, 31), M=(1, 12), Y=(0, 2009))
def birthday(D, M, Y):
    print('birthday = {0}/{1}/{2}'.format(D, M, Y))

persinfo('Robert', 40)
persinfo(age=40, name='Robert')
birthday(1, M=5, Y=1963)
#persinfo('Robert', 150)
#persinfo(age=150, name='Robert')
#birthday(40, M=5, Y=1963)

# Testowanie metod, pozycyjne i po słowach kluczowych
class Person:

    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

        # giveRaise = rangetest(...)

(giveRaise)

    @rangetest(percent=(0.0, 1.0))           # Wartość percent przekazana
po nazwie lub pozycji

```

```

def giveRaise(self, percent):
    self.pay = int(self.pay * (1 + percent))
bob = Person('Robert Zielony', 'programista', 100000)
anna = Person('Anna Czerwona', 'programista', 100000)
bob.giveRaise(.10)
anna.giveRaise(percent=.20)
print(bob.pay, anna.pay)
#bob.giveRaise(1.10)
#bob.giveRaise(percent=1.20)
# Testowanie pominiętych wartości domyślnych: pominięte
@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):
    print(a, b, c, d)
omitargs(1, 2, 3, 4)
omitargs(1, 2, 3)
omitargs(1, 2, 3, d=4)
omitargs(1, d=4)
omitargs(d=4, a=1)
omitargs(1, b=2, d=4)
omitargs(d=8, c=7, a=1)
#omitargs(1, 2, 3, 11)      # Niepoprawne d
#omitargs(1, 2, 11)      # Niepoprawne c
#omitargs(1, 2, 3, d=11)    # Niepoprawne d
#omitargs(11, d=4)      # Niepoprawne a
#omitargs(d=4, a=11)     # Niepoprawne a
#omitargs(1, b=11, d=4)    # Niepoprawne b
#omitargs(d=8, c=7, a=11)   # Niepoprawne a

```

Po wykonaniu powyższego skryptu argumenty niemieszczące się w przedziałach tak jak wcześniej zgłaszały wyjątki, jednak możemy je przekazywać po nazwie bądź pozycji, a pominięte wartości domyślne nie są sprawdzane. Kod działa w Pythonie 2.x oraz 3.x. Warto prześledzić dane wyjściowe i kontynuować testy na własną rękę w celach eksperymentalnych. Kod działa jak wcześniej, jednak jego zakres został rozszerzony.

```
C:\code> python rangetest_test.py
```

```
Robert ma 40 lat
Robert ma 40 lat
Data urodzenia = 1/5/1963
```

```
110000 120000
1 2 3 4
Argument "d" ma wartość domyślną
1 2 3 9
1 2 3 4
Argument "c" ma wartość domyślną
Argument "b" ma wartość domyślną
1 7 8 4
Argument "c" ma wartość domyślną
Argument "b" ma wartość domyślną
1 7 8 4
Argument "c" ma wartość domyślną
1 2 8 4
Argument "b" ma wartość domyślną
1 7 7 8
```

W przypadku błędów w sprawdzaniu otrzymujemy, jak poprzednio, wyjątek (o ile do Pythona nie przekazano argumentu wiersza poleceń -0), jeśli jeden z wierszy testowania metod zostanie wyjęty z komentarza.

```
TypeError: giveRaise argument "percent" nie mieści się w przedziale 0.0..1.0
```

Szczegóły implementacji

Kod tego dekoratora opiera się zarówno na API introspekcji, jak i subtelnych ograniczeniach przekazywania argumentów. By go w pełni uogólnić, powinniśmy spróbować naśladować pełną logikę dopasowywania argumentów Pythona w celu sprawdzenia, które nazwy zostaną przekazane w których trybach, jednak jest to zbyt wysoki stopień skomplikowania jak na nasze narzędzie. Lepiej byłoby, gdybyśmy mogli jakoś argumenty przekazane po nazwie dopasować do zbioru wszystkich oczekiwanych nazw argumentów w celu ustalenia, na jakiej pozycji pojawiają się one w określonym wywołaniu.

Dalsza introspekcja

Okazuje się, że API do introspekcji dostępne dla obiektów funkcji oraz powiązanych z nimi obiektów kodu ma do dyspozycji narzędzie, którego potrzebujemy. API to zostało krótko wprowadzone w rozdziale 19., jednak teraz możemy je wykorzystać. Zbiór oczekiwanych nazw argumentów to po prostu pierwszych N nazw zmiennych dołączonych do obiektu kodu funkcji.

```
# W Pythonie 3.x (oraz 2.x dla zgodności):
>>> def func(a, b, c, e=True, f=None):      # Trzy wymagane i dwa domyślne
    argumenty
    ...
    x = 1
    ...
    y = 2
    ...
    ...
```

```

>>> code = func.__code__                                # Obiekt kodu obiektu funkcji
>>> code.co_nlocals
6
>>> code.co_varnames                                 # Nazwy wszystkich zmiennych
lokalnych
('a', 'b', 'c', 'e', 'f', 'x', 'y')
>>> code.co_varnames[:code.co_argcount]      # Pierwsze N zmiennych lokalnych
to oczekiwane argumenty
('a', 'b', 'c', 'e', 'f')

```

Jak zwykle: za pomocą argumentów, które w obiekcie pośredniczącym są opatrzone gwiazdkami, można odczytać dowolną liczbę argumentów i dopasować je do oczekiwanych argumentów:

```

>>> def catcher(*pargs, **kargs): print('%s, %s' % (pargs, kargs))
>>> catcher(1, 2, 3, 4, 5)
(1, 2, 3, 4, 5), {}
>>> catcher(1, 2, c=3, d=4, e=5)                  # Argumenty wywoływanej funkcji
(1, 2), {'d': 4, 'e': 5, 'c': 3}

```

To samo API dostępne jest również w starszych wersjach Pythona, jednak atrybut `func.__code__` zapisywany jest jako `func.func_code` w wersji 2.5 oraz wcześniejszych (nowszy atrybut `__code__` jest również dostępny w wersji 2.6 z uwagi na przenośność). Więcej informacji można otrzymać po wykonaniu wywołania funkcji `dir` na obiektach funkcji oraz kodu. Kod taki jak niżej działa w wersjach 2.5 i starszych, choć sama wartość `sys.version_info` nie jest uniwersalna — w nowych wersjach Pythona jest to nazwana krotką. Natomiast w starszych i nowszych wersjach można używać przesunięć:

```

>>> import sys                                     # Dla zgodności z poprzednimi
wersjami
>>> sys.version_info                               # [0] to numer dużego wydania
(3, 3, 0, 'final', 0)
>>> code = func.__code__ if sys.version_info[0] == 3 else func.func_code

```

Założenia dotyczące argumentów

Gdy mamy zbiór oczekiwanych nazw argumentów, rozwiązanie opiera się na dwóch ograniczeniach w zakresie *kolejności* ich przekazywania, narzuconych przez Pythona (w wersjach 2.x oraz 3.x nadal są one obowiązujące):

- W wywołaniu wszystkie argumenty pozycyjne pojawiają się przed wszystkimi argumentami ze słowami kluczowymi.
- W instrukcji `def` wszystkie argumenty bez wartości domyślnej pojawiają się przed wszystkimi argumentami z wartościami domyślnymi.

Oznacza to, że argument niebędący słowem kluczowym nie może w wywołaniu pojawić się po argumencie ze słowem kluczowym, natomiast argument bez wartości domyślnej nie może się pojawić po argumencie z wartością domyślną w *definicji* funkcji. Cała składnia *nazwa=wartość* musi się w obu miejscach pojawiać po wszystkich zwykłych argumentach typu *nazwa*. Ponadto, jak już wiemy, Python dopasowuje wartości argumentów podanych za pomocą pozycji do ich

nazw w kolejności od lewej do prawej, przez co wartości te są zawsze przypisywane do nazw znajdujących się w nagłówkach najbliżej lewej strony. Natomiast słowa kluczowe dopasowywane są według nazw, a dany argument może mieć tylko jedną wartość.

W celu uproszczenia sobie pracy możemy także założyć, że wywołanie jest ogólnie *poprawne*, to znaczy wszystkie argumenty albo otrzymają wartości (po nazwie bądź pozycji), albo zostaną celowo pominięte w celu pobrania wartości domyślnych. Takie założenie niekoniecznie będzie prawdziwe, ponieważ funkcja nie zostaje jeszcze wywołana, zanim logika opakowująca sprawdzi poprawność. Wywołanie może nadal się nie powieść później, po uruchomieniu za pośrednictwem warstwy opakowującej, z uwagi na niepoprawne przekazanie argumentów. Dopóki jednak nie będzie to powodowało jeszcze większego błędu w warstwie opakowującej, będziemy mogli pracować nad szczegółami poprawności wywołania. Jest to dla nas pomocne, ponieważ sprawdzanie poprawności wywołań przed ich wykonaniem wymagałoby pełnej emulacji algorytmu dopasowywania argumentów Pythona — co jest zbyt skomplikowaną procedurą jak na nasze narzędzie.

Algorytm dopasowywania

Po przyjęciu powyższych założeń i poznaniu ograniczeń możemy teraz w naszym algorytmie zarówno pozwolić na użycie argumentów ze słowami kluczowymi, jak i pomijanie argumentów z wartościami domyślnymi w wywołaniu. Kiedy wywołanie zostaje przechwycone, możemy poczynić następujące założenia:

1. Niech N oznacza liczbę argumentów pozycyjnych równą długości krotki `*pargs`.
2. Wszystkie N przekazanych argumentów pozycyjnych z `*pargs` muszą odpowiadać pierwszym N oczekiwanych argumentów pozyskanych z obiektu kodu funkcji. Jest to prawdziwe zgodnie z przedstawionymi wcześniej regułami kolejności wywołań Pythona, ponieważ wszystkie argumenty pozycyjne pojawiają się przed wszystkimi argumentami ze słowami kluczowymi.
3. W celu uzyskania nazw argumentów przekazanych po pozycji musimy wykonać wycinek z listy wszystkich oczekiwanych argumentów aż do długości N krotki pozycyjnej `*pargs`.
4. Wszystkie argumenty po pierwszych N oczekiwanych argumentach zostały albo przekazane po słowie kluczowym, albo pominięte w czasie wywołania i przyjmują wartości domyślne.
5. Każda nazwa argumentu sprawdzanego przez dekorator jest przetwarzana w następujący sposób:
 - a. Jeśli znajduje się w `**kargs`, to znaczy, że argument został przekazany za pomocą nazwy i jego wartość jest odczytywana poprzez indeksowanie `**kargs`.
 - b. Jeśli znajduje się ona w pierwszych N oczekiwanych argumentach to znaczy, że została przekazany po pozycji (w którym to przypadku jego względna pozycja na liście oczekiwanych argumentów określa jego względną pozycję w krotce `*pargs`).
 - c. W przeciwnym razie możemy założyć, że argument został pominięty w wywołaniu, ma wartość domyślną i nie musi być sprawdzany.

Innymi słowy, możemy pominąć testy argumentów, które zostały pominięte w wywołaniu, zakładając, że pierwszych N faktycznie przekazanych argumentów pozycyjnych z `*pargs` musi odpowiadać pierwszym N nazw argumentów z listy wszystkich oczekiwanych argumentów, a wszystkie pozostałe albo musiały być przekazane po słowach kluczowych (i tym samym znajdować się w `**kargs`), albo przybierają wartości domyślne. W tym rozwiązaniu dekorator

po prostu pominie wszelkie argumenty do sprawdzenia pominięte pomiędzy znajdującym się najbardziej na prawo argumentem pozycyjnym a znajdującym się najbardziej na lewo argumentem ze słowem kluczowym, pomiędzy argumentami ze słowami kluczowymi lub ogólnie po argumencie pozycyjnym znajdującym się najbardziej na prawo. By przekonać się, w jaki sposób odbywa się to w kodzie, warto prześledzić kod dekoratora oraz jego skryptu testowego.

Znane problemy

Choć nasze narzędzie sprawdzające przedziały działa zgodnie z planem, ma trzy mankamenty: nie wykrywa niepoprawnych wywołań, nie obsługuje niektórych sygnatur dowolnych argumentów i nie obsługuje zagnieżdżonych dekoratorów. Rozwiązaniem może być napisanie rozszerzenia lub zastosowanie zupełnie innego podejścia. Poniżej przedstawiony jest krótki opis poszczególnych problemów.

Niepoprawne wywołania

Po pierwsze, jak wspomniano wcześniej, wywołania oryginalnej funkcji, które nie są poprawne, nadal nie powiodą się w ostatecznej wersji dekoratora. Przykładowooba poniższe wywołania zwracają wyjątki:

```
omitargs()  
omitargs(d=8, c=7, b=6)
```

Nie powiodą się one jednak tylko tam, gdzie próbujemy wywołać oryginalną funkcję — na końcu obiektu opakowującego. Choć moglibyśmy próbować imitować mechanizm dopasowywania argumentów Pythona w celu uniknięcia tej sytuacji, nie za bardzo jest powód, by to robić — ponieważ wywołanie na tym etapie i tak by się nie powiodło, możemy równie dobrze pozwolić własnej logice dopasowywania argumentów Pythona na wykrycie tego problemu za nas.

Dowolne argumenty

Po drugie, choć nasza ostateczna wersja kodu obsługuje argumenty pozycyjne, argumenty ze słowami kluczowymi oraz pominięte wartości domyślne, nadal nie robi nic z `*args` oraz `**args`, które mogą zostać użyte w udekorowanej funkcji przyjmującej *dowolną liczbę* argumentów. Najprawdopodobniej nie będziemy się tym jednak musieli przejmować w naszej sytuacji:

- Jeśli dodatkowy argument *ze słowem kluczowym* zostanie przekazany, jego nazwa pokaże się w słowniku `**kargs` i zostanie on normalnie przetestowany, o ile zostanie wspomniany w dekoratorze.
- Jeśli dodatkowy argument *ze słowem kluczowym nie* zostanie przekazany, jego nazwy nie znajdziemy ani w słowniku `**kargs`, ani w wycinku oczekiwanych argumentów pozycyjnych, dlatego nie zostanie on sprawdzony. Zostanie potraktowany tak, jakby miał mieć wartość domyślną, nawet jeśli tak naprawdę jest to dodatkowy argument opcjonalny.
- Jeśli przekazany zostanie dodatkowy argument *pozycyjny*, i tak nie można się do niego w żaden sposób odwołać w dekoratorze — jego nazwy nie będzie ani w słowniku `**kargs`, ani w wycinku listy oczekiwanych argumentów, dlatego zostanie po prostu pominięty. Ponieważ takie argumenty nie są wymienione w definicji funkcji, nie można odwzorować nazwy podanej do dekoratora z powrotem na oczekwaną względową pozycję.

Innymi słowy, w obecnej postaci kod obsługuje sprawdzanie dowolnych argumentów ze słowami kluczowymi po nazwie, ale już nie dowolnych argumentów pozycyjnych, które pozostały bez nazwy i tym samym nie mają ustalonej pozycji w sygnaturze argumentu funkcji. W przypadku interfejsu API obiektu funkcyjnego efekt użycia tych narzędzi w udekorowanej funkcji jest następujący:

```
>>> def func(*kargs, **pargs): pass
```

```
>>> code = func.__code__
>>> code.co_nlocals, code.co_varnames
(2, ('kargs', 'pargs'))
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(0, ())
>>> def func(a, b, *kargs, **pargs): pass
>>> code = func.__code__
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(2, ('a', 'b'))
```

Ponieważ argumenty z gwiazdkami są widoczne jako zmienne lokalne, a *nie* jako oczekiwane argumenty, nie mają one wpływu na nasz algorytm dopasowujący — poprzedzające je nazwy w nagłówku funkcji można weryfikować w zwykły sposób, ale nie można przekazywać żadnych dodatkowych argumentów pozycyjnych. Co do zasady moglibyśmy rozszerzyć interfejs dekoratora, tak by obsługiwał on *pargs w udekorowanej funkcji na potrzeby rzadkich przypadków, w których może się to przydać (na przykład specjalna nazwa argumentu z testem do zastosowania do wszystkich argumentów w krotce *pargs obiektu opakowującego wykraczających poza długość listy oczekiwanych argumentów), ale nie będziemy się tutaj zajmować takim rozszerzeniem.

Zagnieżdżone dekoratory

Trzeci i najbardziej subtelny problem polega na tym, że opisany sposób kodowania nie pozwala na pełne zagnieżdżanie dekoratorów i łączenie operacji. Ponieważ w definicjach funkcji argumenty są analizowane za pomocą nazw, a nazwy użyte w funkcji pośredniczącej zwracane przez zagnieżdżony dekorator nie odpowiadają nazwom argumentów oryginalnej funkcji lub dekoratora, więc tryb zagnieżdżania nie jest w pełni obsługiwany.

Z technicznego punktu widzenia podczas zagnieżdżania w pełni wykonywane są tylko najgłębiej zagnieżdżone weryfikacje. Na wszystkich pozostałych poziomach zagnieżdżenia testy są wykonywane tylko na argumentach przekazywanych za pomocą słów kluczowych. Prześledźmy kod, aby dowiedzieć się, dlaczego tak jest. Ponieważ sygnatura funkcji pośredniczącej `onCall` nie zawiera nazwanych argumentów pozycyjnych, wszystkie argumenty przeznaczone do sprawdzenia, przekazane za pomocą pozycji, są traktowane, jakby nie były użyte i otrzymały domyślne wartości, dlatego ostatecznie są pomijane.

To może być cecha właściwa zastosowanemu rozwiązaniu. Funkcje pośredniczące zmieniają sygnatury nazw argumentów na ich poziomach, przez co nie ma możliwości bezpośredniego wiązania nazw w argumentach dekoratora z pozycjami przekazywanymi w sekwencjach argumentów. Jeżeli stosowane są funkcje pośredniczące, wtedy nazwy argumentów ostatecznie mają zastosowanie tylko w przypadku słów kluczowych. Natomiast wstępne rozwiązanie oparte na pozycjach argumentów może lepiej obsługiwać funkcje pośredniczące, ale nie obsługuje w pełni słów kluczowych.

Zamiast implementowania zagnieżdżania w rozwiązaniu quizu na końcu rozdziału uogólnimy dekorator tak, aby pojedynczo realizował różne rodzaje weryfikacji. Dodatkowo pokazane tam będą przykłady ograniczeń zagnieżdżania. Ponieważ jednak wyczerpaliśmy już miejsce przeznaczone na ten przykład, osoby, które mają ochotę zająć się takimi ulepszeniami, oficjalnie kierujemy do krainy ćwiczeń sugerowanych.

Argumenty dekoratora a adnotacje funkcji

Co ciekawe, opcja adnotacji funkcji wprowadzona w wersjach Pythona 3.0 i nowszych, mogłyby stanowić alternatywę dla argumentów dekoratora wykorzystywanych w naszym przykładzie do określania testów przedziałów. Jak wiemy z rozdziału 19., adnotacje pozwalają nam wiązać wyrażenia z argumentami i zwracanymi wartościami poprzez zapisanie ich w kodzie w samym wierszu nagłówka instrukcji `def`. Python zbiera adnotacje w słownik i doda je do opisanej w ten sposób funkcji.

Moglibyśmy wykorzystać to w naszym przykładzie do zapisania granic przedziałów w wierszu nagłówka zamiast w argumentach dekoratora. Nadal potrzebowalibyśmy dekoratora funkcji do opakowania funkcji w celu przechwycenia późniejszych wywołań, jednak zamienilibyśmy następującą składnię argumentów dekoratora:

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                      # func = rangetest(...)(func)
    print(a + b + c)
```

na poniższą składnię adnotacji:

```
@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)
```

Oznacza to, że granice przedziałów zostałyby przesunięte do samej funkcji, a nie pozostałyby w kodzie poza nią. Poniższy skrypt ilustruje strukturę wynikowych dekoratorów w obu rozwiązaniach, w niekompletnym szkielecie kodu. Wzorzec z argumentami dekoratorów pochodzi z zaprezentowanego wcześniej pełnego rozwiązania. Alternatywa z adnotacjami wymaga o jeden poziom zagnieżdżenia mniej, ponieważ nie musi zachowywać argumentów dekoratorów.

```
# Wykorzystanie argumentów dekoratora (wersje 3.x i 2.x)

def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks: pass          # Tutaj dodanie kodu
sprawdzającego
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                      # func = rangetest(...)(func)
    print(a + b + c)
    func(1, 2, c=3)                      # Wykonuje onCall, argchecks w
zakresie

# Wykorzystanie adnotacji funkcji (tylko wersja 3.x)

def rangetest(func):
```

```

def onCall(*pargs, **kargs):
    argchecks = func.__annotations__
    print(argchecks)
    for check in argchecks:
        pass          # Tutaj dodanie kodu sprawdzającego
    return func(*pargs, **kargs)

return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):           # func = rangetest(func)
    print(a + b + c)
func(1, 2, c=3)                                # Wykonuje onCall, adnotacje w
                                                funkcji

```

Po wykonaniu oba rozwiązania mają dostęp do tych samych informacji testu sprawdzającego, jednak w innych formach. Informacje z wersji z argumentami dekoratora zachowywane są w argumencie w zakresie zawierającym, natomiast informacje z wersji z adnotacją zachowywane są w atrybutie samej funkcji. Natomiast w wersji 3.x ze względu na zastosowanie adnotacji funkcji uzyskujemy następujący wynik:

```

C:\code> py -3 decoargs-vs-annotation.py
{'a': (1, 5), 'c': (0.0, 1.0)}
6
{'a': (1, 5), 'c': (0.0, 1.0)}
6

```

Utworzenie reszty wersji opartej na adnotacji pozostawiam jako sugerowane ćwiczenie; jej kod byłby identyczny z kodem zaprezentowanego wcześniej pełnego rozwiązania, ponieważ informacje z testów przedziałów znajdują się po prostu w funkcji zamiast w zakresie zawierającym. Tak naprawdę rozwiązanie to daje nam inny interfejs użytkownika dla naszego narzędzia — nadal, jak wcześniej, będzie ono musiało dopasowywać nazwy argumentów do nazw oczekiwanych argumentów w celu uzyskania ich względnych pozycji.

Właściwie użycie adnotacji w miejsce argumentów dekoratora w tym przykładzie tak naprawdę ogranicza jego przydatność. Po pierwsze, adnotacje działają jedynie w Pythonie 3.x, dlatego wersja 2.x przestaje być obsługiwana. Z kolei dekoratory funkcji z argumentami działają w obu wersjach Pythona.

Co jednak bardziej istotne, przenosząc specyfikację sprawdzania poprawności do nagłówka, tak naprawdę ograniczamy funkcję do jednej roli. Ponieważ adnotacja pozwala nam na zapisanie w kodzie tylko jednego wyrażenia na argument, może pełnić tylko jeden cel. Przykładowo nie możemy wykorzystać adnotacji testów przedziałów do żadnej innej roli.

Z kolei, ponieważ argumenty dekoratora zapisywane są w kodzie poza samą funkcję, nie tylko łatwiej jest je usunąć, ale są one również bardziej ogólne. Kod samej funkcji nie wymusza jednego celu dekoracji. Tak naprawdę dzięki zagnieżdżeniu dekoratorów z argumentami możemy zastosować większą liczbę kroków rozszerzających do tej samej funkcji. Adnotacja w sposób bezpośredni obsługuje tylko jeden krok. Po zastosowaniu argumentów dekoratora sama funkcja zachowuje także prostszy, normalny wygląd.

Mimo to, jeśli mamy na myśli jeden konkretny cel i możemy się zdecydować na obsługę jedynie Pythona 3.x, wybór pomiędzy adnotacjami a argumentami dekoratora jest w dużej mierze subiektywny i sprowadza się do stylistyki. Jak to się często zdarza w życiu, adnotacje jednej osoby dla drugiej mogą być składniowym bałaganem...

Inne zastosowania — sprawdzanie typów (skoro nalegamy!)

Wzorzec kodu, na jakim skończyliśmy przetwarzanie argumentów w dekoratorach, można również zastosować w innych kontekstach. Sprawdzanie typów danych argumentów w czasie programowania jest na przykład prostym rozszerzeniem kodu.

```
def typetest(**argchecks):
    def onDecorator(func):
        ...
        def onCall(*pargs, **kargs):
            positionals = list(allargs)[:len(pargs)]
            for (argname, type) in argchecks.items():
                if argname in kargs:
                    if not isinstance(kargs[argname], type):
                        ...
                        raise TypeError(errmsg)
                elif argname in positionals:
                    position = positionals.index(argname)
                    if not isinstance(pargs[position], type):
                        ...
                        raise TypeError(errmsg)
                else:
                    # Zakładamy, że nie przekazano:
                    wartości domyślne
                    return func(*pargs, **kargs)
            return onCall
        return onDecorator
    @typetest(a=int, c=float)
    def func(a, b, c, d):               # func = typetest(...)(func)
        ...
        func(1, 2, 3.0, 4)             # OK
        func('mielonka', 2, 99, 4)     # Poprawnie uruchamia wyjątek
```

Wykorzystanie adnotacji funkcji zamiast argumentów dekoratora dla takiego dekoratora, zgodnie z opisem z poprzedniego rozdziału, sprawiłoby, że kod w jeszcze większym stopniu przypominałby deklaracje typów z innych języków programowania.

```
@typetest  
def func(a: int, b, c: float, d):           # func = typetest(func)  
    ...  
        # Ach! ...
```

W ten sposób jednak niebezpiecznie blisko pojawia się „żółta kartka”. Czego jednak każdy powinien był się nauczyć z niniejszej książki, ta akurat rola jest ogólnie złym pomysłem w działającym kodzie, nie mówiąc o tym, że zupełnie nie jest w pythonowym stylu (często jest zresztą traktowana jako symptom pierwszych prób użycia Pythona przez byłego programistę języka C++).

Sprawdzanie typów ogranicza działanie funkcji do określonych typów, zamiast pozwalać jej działać na dowolnych typach ze zgodnymi *interfejsami*. W rezultacie ogranicza nasz kod i łamie jego *elastyczność*. Z drugiej strony, od każdej reguły są wyjątki — sprawdzanie typów może się przydać w wyizolowanych przypadkach w trakcie debugowania i pisania interfejsów dla kodu napisanego w bardziej restrykcyjnym języku, takim jak C++.

Ten ogólny wzorzec przetwarzania argumentów może także mieć zastosowanie w różnych mniej kontrowersyjnych rolach. Tak naprawdę moglibyśmy nawet uogólnić to narzędzie jeszcze bardziej, przekazując *funkcję sprawdzającą*, podobnie jak zrobiliśmy to wcześniej w dekoracjach atrybutów jako publicznych. Pojedyncza kopia tego typu kodu wystarczyłaby do sprawdzania przedziałów, typów i temu podobnych zastosowań. Takie uogólnienie zawarte jest w quizie na końcu rozdziału, zatem to rozszerzenie potraktujmy tutaj jako przerwanie akcji w kulminacyjnym momencie.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy dekoratory w odmianach związanych z funkcjami oraz klasami. Zgodnie z tym, czego się nauczyliśmy, dekoratory są sposobem wstawiania kodu, który będzie wykonywany automatycznie przy definiowaniu funkcji bądź klas. Kiedy dekorator jest wykorzystywany, Python ponownie dowiązuje nazwę funkcji bądź klasy do zwracanego przez niego obiektu wywoływalnego. Ten punkt zaczepienia pozwala nam dodać warstwę logiki opakowującej do wywołań funkcji oraz wywołań tworzących instancje klas w celu zarządzania funkcjami iinstancjami. Jak również widzieliśmy, ten sam efekt można osiągnąć za pomocą funkcji zarządzających oraz ponownego dowiązywania nazw, jednak dekoratory udostępniają rozwiązanie bardziej jednorodne i jawne.

Dowiedzieliśmy się również, że dekoratory klas można wykorzystywać do zarządzania samymi klasami, a nie tylko ich instancjami. Ponieważ ta funkcjonalność pokrywa się z metaklasami, tematem kolejnego i ostatniego technicznego rozdziału, dalsza część tej historii znajduje się nieco dalej. Najpierw jednak należy wykonać quiz kończący rozdział. Ponieważ rozdział ten skupiał się w przeważającej mierze na większych przykładach, w quizie Czytelnik zostanie poproszony o zmodyfikowanie części kodu w celu sprawdzenia go. Oryginalne wersje kodów znajdują się w załączonych do książki plikach (patrz wskazówki w przedmowie). Jeżeli czas nagli, można przestudiować modyfikacje opisane w odpowiedziach — programowanie polega nie tylko na pisaniu kodu, ale też na czytaniu go.

Sprawdź swoją wiedzę — quiz

1. *Dekoratory metod*: jak wspomniano w jednej ze wskazówek rozdziału, napisany w podrozdziale „Dodawanie argumentów dekoratora” i zawarty w pliku *timerdeco2.py* dekorator z argumentami mierzący czas wykonania funkcji można zastosować jedynie do prostych *funkcji*, ponieważ wykorzystuje on zagnieżdżoną klasę z metodą przeciążania operatora `__call__` przechwytyującą wywołania. Struktura ta nie działa jednak dla *metod* klas, ponieważ do `self` przekazywana jest instancja dekoratora, a nie instancja klasy podmiotowej.

Należy przepisać ten dekorator w taki sposób, by można go było zastosować zarówno do prostych funkcji, jak i metod klas, a następnie przetestować na funkcjach i metodach. (Uwaga: wskazówkę należy szukać w podrozdziale zatytułowanym „Uwagi na temat klas I — dekorowanie metod klas”). Warto zauważać, że można skorzystać z przypisywania *atributów* obiektów funkcji w celu śledzenia całkowitego czasu, ponieważ nie będziemy mieli zagnieżdżonej klasy dla celów zachowywania stanu i nie możemy uzyskać dostępu do zmiennych nielokalnych spoza kodu dekoratora. Dodatkowe punkty będą przyznane za napisanie uniwersalnego dekoratora działającego w wersjach Pythona 3.x i 2.x.

2. *Dekoratory klas*: dekoratory klas `Public` i `Private`, napisane w niniejszym rozdziale (zawarte w pliku *access2.py*), dodają pewne obciążenie związane z każdym pobraniem atrybutu w udekorowanej klasie. Choć moglibyśmy po prostu usunąć wiersz z dekoracją @ w celu zyskania szybkości kodu, możemy również rozszerzyć sam dekorator w taki sposób, by sprawdzał przełącznik `_debug_` i nie wykonywał opakowywania wtedy, gdy opcja -0 Pythona została przekazana w wierszu poleceń (tak samo, jak robiliśmy to w przypadku dekoratorów sprawdzających przedziały argumentów). W ten sposób możemy zwiększyć szybkość działania programu bez zmiany jego źródła za pomocą samych argumentów wiersza polecen (`python -0 main.py...`). Możemy również wykorzystać jedną z opisanych technik wykorzystujących nadzędne klasy mieszane i przechwytywać kilka wbudowanych operacji w Pythonie 3.x. Należy napisać kod takiego rozszerzenia oraz odpowiedniego testu.
3. *Uniwersalne weryfikowanie argumentów*: dekoratory funkcji i metod, zawarte w pliku *rangetest.py*, sprawdzają, czy wartości argumentów mieszą się w dopuszczalnych zakresach. Jak się jednak przekonaliśmy, ten sam wzorzec można stosować do innych celów, na przykład do sprawdzania typów argumentów. Zadanie polega na uogólnieniu dekoratora sprawdzającego zakresy, tak aby można było jeden kod wykorzystywać do weryfikowania poprawności argumentów według różnych kryteriów. Biorąc pod uwagę przedstawiony tu sposób kodowania, najprostszym rozwiązaniem może być przekazywanie funkcji, jednak w kontekście bardziej obiektowego programowania podobne uogólnienie można osiągnąć również za pomocą podklas zawierających wymagane metody.

Sprawdź swoją wiedzę — odpowiedzi

1. Poniżej znajduje się jeden ze sposobów zapisania w kodzie rozwiązania, a także jego wynik (choć z wykorzystaniem metod klas, które wykonywane są zbyt szybko, by można było zmierzyć czas ich działania). Sztuczka polega na zastąpieniu zagnieżdżonych klas *funkcjami zagnieżdżonymi*, tak by argument `self` nie był instancją dekoratora, a także przypisaniu całkowitego czasu do samej funkcji dekoratora, tak by można go było pobrać później za pomocą oryginalnej, dowiezanej ponownie nazwy (więcej szczegółów na ten temat można znaleźć w podrozdziale „Możliwości w zakresie zachowania informacji o stanie” — funkcje obsługują

dodłączanie dowolnych atrybutów, a nazwa funkcji jest w tym kontekście referencją do zakresu zawierającego). Kod można dodatkowo rozszerzyć o rejestrowanie najlepszego (*minimalnego*) czasu wykonania obok całkowitego, tak jak w rozdziale 21. w przykładach mierzenia czasu.

```
"""
```

Plik timerdeco.py (wersje 3.x i 2.x)

Dekorator mierzący czas wykonania zarówno funkcji, jak i metody.

```
"""
```

```
import time
```

```
def timer(label='', trace=True): # Dla argumentów dekoratora: zachowanie argumentów
```

```
    def onDecorator(func): # Dla dekoracji @: zachowanie udekorowanej funkcji
```

```
        def onCall(*args, **kargs): # W momencie wywołania: wywołanie oryginału
```

```
            start = time.clock() # Stan to zakresy – atrybuty funkcji
```

```
            result = func(*args, **kargs)
```

```
            elapsed = time.clock() - start
```

```
            onCall.alltime += elapsed
```

```
            if trace:
```

```
                format = '%s%s: %.5f, %.5f'
```

```
                values = (label, func.__name__, elapsed, onCall.alltime)
```

```
                print(format % values)
```

```
            return result
```

```
        onCall.alltime = 0
```

```
        return onCall
```

```
    return onDecorator
```

Test został zakodowany w osobnym pliku, aby można było łatwiej korzystać z dekoratora.

```
"""
```

Plik timerdeco-test.py

```
"""
```

```
from __future__ import print_function # Wersja 2.x
```

```
from timerdeco import timer
```

```
import sys
```

```
force = list if sys.version_info[0] == 3 else (lambda X: X)
```

```

print('-----')
# Przetestowanie na funkcjach
@timer(trace=True, label='[CCC]==>')
def listcomp(N):
    # Jak listcomp
    = timer(...)(listcomp)

    return [x * 2 for x in range(N)] # listcomp(...)

wywołuje onCancel

@timer(' [MMM]==>')
def mapcall(N):
    return list(map((lambda x: x * 2), range(N))) # list() dla
widoków z Pythona 3.x

for func in (listcomp, mapcall):

    result = func(5) # Czas dla tego wywołania,
wszystkich wywołań, zwracana wartość

    func(5000000)
    print(result)

    print('allTime = %s\n' % func.alltime) # Całkowity
czas wszystkich wywołań

print('-----')
# Przetestowanie na metodach

class Person:

    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @timer()
    def giveRaise(self, percent): # giveRaise =
        timer()(giveRaise)

        self.pay *= (1.0 + percent) # tracer
        pamięta giveRaise

    @timer(label='**')

    def lastName(self): # lastName =
        timer(...)(lastName)

        return self.name.split()[-1] # Całkowity
        czas per klasa, nie instancja

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
bob.giveRaise(.10)

```

```

anna.giveRaise(.20)                                # Wykonuje
onCall(anna, .10)

print(bob.pay, anna.pay)

print(bob.lastName(), anna.lastName())            # Wykonuje
onCall(bob), pamięta lastName

print('%.5f      %.5f'      %           (Person.giveRaise.alltime,
Person.lastName.alltime))

```

Jeżeli wszystko działa zgodnie z oczekiwaniami, wyniki uzyskane w wersjach Pythona 3.x i 2.x będą takie jak niżej. Inne mogą być jedynie wartości pomiaru czasu, w zależności od wersji Pythona i szybkości komputera:

```

c:\code> py -3 timerdeco-test.py
-----
[CCC]==>listcomp: 0.00001, 0.00001
[CCC]==>listcomp: 0.57930, 0.57930
[0, 2, 4, 6, 8]
allTime = 0.5793010457092784
[MMM]==>mapcall: 0.00002, 0.00002
[MMM]==>mapcall: 1.08609, 1.08611
[0, 2, 4, 6, 8]
allTime = 1.0861149923442373
-----
giveRaise: 0.00001, 0.00001
giveRaise: 0.00001, 0.00002
55000 120000
**lastName: 0.00001, 0.00001
**lastName: 0.00000, 0.00001
Zielony Czerwona
0.00001 0.00001

```

- Poniższe rozwiązanie odpowiada na drugie pytanie. Kod został rozszerzony, tak by zwracać oryginalną klasę w trybie zoptymalizowanym (-O), aby dostęp do atrybutów nie wpływał negatywnie na szybkość działania programu. Tak naprawdę jedyną czynnością, którą wykonałem, było dodanie instrukcji sprawdzających tryb debugowania, a także dalsza indentacja klasy w prawą stronę.

....

Plik access.py (wersje 3.x i 2.x)

Dekorator klasy z prywatnymi i publicznymi deklaracjami atrybutów.

Kontroluje dostęp z zewnątrz do atrybutów zapisanych w instancji lub odziedziczonych

przez nią w dowolny sposób z innych klas.

Dekorator `Private` deklaruje nazwy atrybutów, których nie można odczytywać ani

przypisywać im wartości poza udekorowaną klasą. Dekorator Public deklaruje wszystkie

nazwy, na których można wykonywać powyższe operacje.

Uwaga: w wersji 3.x przechwytywane są wyłącznie te wbudowane operacje, które są

zakodowane w klasie `BuiltinMixins` (jej kod można rozszerzyć). Dlatego dekorator `Public`

zakodowany w takiej postaci jak tutaj może być mniej przydatny przy przeciążaniu

operatorów niż dekorator Private.

三

```
from access_builtins import BuiltinsMixin           # Częściowa obsługa!
```

```
traceMe = False
```

```
def trace(*args):
```

```
if traceMe: print('[' + ' '.join(map(str, args)) + ']')
```

```
def accessControl(failIf):
```

```
def onDecorator(aClass):
```

if not debug

return aClass

else:

```
class onInstance:
```

```
def __init__(self, *args, **kwargs):
```

```
self.wrapped = aClass(*args, **kwargs)
```

```
def __getattr__(self, attr):
```

```
trace('pobranie:', attr)
```

```
if failIf(attr):
```

```
raise TypeError('pobranie atrybutu prywatnego: ')
```

else:

```
return getattr(self.wrapped, attr)
```

```
def setattr (self, attr, value):
```

```
trace('ustawienie:', attr, value)
```

```
if attr == 'onInstance wrapped':
```

```

        self.__dict__[attr] = value
    elif failIf(attr):
        raise TypeError('modyfikacja atrybutu
prywatnego: ' + attr)
    else:
        setattr(self.__wrapped__, attr, value)
    return onInstance
return onDecorator
def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))
def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in
attributes))

```

Do ponownego zdefiniowania metod przeciążających operatory w opakowującej klasie została zastosowana technika wykorzystująca klasy mieszane. Dzięki temu w wersji 3.x wbudowane operacje są poprawnie delegowane do klas wykorzystujących te metody. Zakodowany w ten sposób pośrednik jest typową klasą w wersji 2.x, która kieruje operacją przez metodę `__getattr__`. Jednak w wersji 3.x jest to klasa w nowym stylu, która tego nie robi. Wykorzystana tu klasa mieszana wymaga wymienienia tych metod w dekoratorach `Public`. Wcześniej zostały opisane alternatywne rozwiązania, które tego nie wymagają (ale też nie pozwalają definiować prywatnych operacji wbudowanych) i rozszerzają tę klasę odpowiednio do potrzeb:

```

"""
Plik access_builtin.py

Przekierowanie niektórych wbudowanych operacji z powrotem do klasy
pośredniczącej,
dzięki czemu działają tak samo w wersji 3.x, jak wywoływane
bezośrednio po nazwach
i w domyślnych klasach w wersji 2.x. Kod w miarę potrzeby można
rozszerzyć tak, aby
obejmował inne nazwy __X__ wykorzystywane w obiektach
pośredniczących.

"""

class BuiltinsMixin:
    def reroute(self, attr, *args, **kargs):
        return self.__class__.__getattr__(self, attr)(*args,
**kargs)
    def __add__(self, other):
        return self.reroute('__add__', other)
    def __str__(self):

```

```

        return self.reroute('__str__')
    def __getitem__(self, index):
        return self.reroute('__getitem__', index)
    def __call__(self, *args, **kargs):
        return self.reroute('__call__', *args, **kargs)
    # Plus inne operatory wykorzystywane tylko w wersji 3.x w
    opakowanych obiektach

```

Tutaj również samotestujący kod został wyodrębniony do osobnego pliku, dzięki czemu dekorator można dowolnie importować bez uruchamiania testu, stosowania atrybutu `_name_` i wcięcia kodu:

```

"""
Plik: access-test.py
Kod testu umieszczony w osobnym pliku, aby można było wielokrotnie
korzystać z dekoratora.

"""

import sys
from access import Private, Public
print('-----')
# Test 1: publiczne nazwy
@Private('age')                                # Person = Private('age')
(Person)
class Person:                                    # Person = onInstance ze stanem
    def __init__(self, name, age):
        self.name = name
        self.age = age                         # Dostęp z wewnętrz działa
normalnie
X = Person('Robert', 40)
print(X.name)                                    # Dostęp z zewnątrz jest
sprawdzany
X.name = 'Anna'
print(X.name)
X + 10
print(X)
try: t = X.age                                    # NIE POWIEDZIE SIĘ, o ile nie
mamy "python -O"
except: print(sys.exc_info()[1])
try: X.age = 999        # Tak samo

```

```

except: print(sys.exc_info()[1])
print('-----')
# Test 2: nazwy prywatne
# Operatory zdefiniowane w klasie BuiltinMixin muszą być
nieprywatne lub publiczne
@Public('name', '__add__', '__str__', '__coerce__')
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __add__(self, N):
        self.age += N # Wbudowane operatory przechwytywane z
wykorzystaniem klas mieszanych w wersji 3.x
    def __str__(self):
        return '%s: %s' % (self.name, self.age)
X = Person('bob', 40)                      # X to onInstance
print(X.name)                               # onInstance osadza Person
X.name = 'Anna'
print(X.name)
X + 10
print(X)
try: t = X.age                                # NIE POWIEDZIE SIĘ, o ile nie
mamy "python -O"
except: print(sys.exc_info()[1])
try: X.age = 999      # Tak samo
except: print(sys.exc_info()[1])

```

Na koniec, jeżeli wszystko działa zgodnie z oczekiwaniami, wyniki uzyskane w wersjach Pythona 3.x i 2.x będą takie jak niżej. Ten sam kod został wykorzystany do przetestowania tej samej klasy udekorowanej najpierw jako prywatna, a potem jako publiczna.

```

c:\code> py -3 access-test.py
-----
Robert
Anna
Anna: 50
pobranie atrybutu prywatnego: age
modyfikacja atrybutu prywatnego: age

```

```
-----  
Robert  
Anna  
Anna: 50  
    pobranie atrybutu prywatnego: age  
    modyfikacja atrybutu prywatnego: age  
c:\code> py -3 -O access-test.py      # Zablokowanie czterech  
powyższych komunikatów o błędach dostępu
```

3. Poniżej przedstawiony jest do samodzielnego przestudiowania uogólniony dekorator weryfikujący argumenty. Wykorzystana jest w nim funkcja weryfikująca, której przekazywane są kryteria sprawdzające zakodowane w argumencie dekoratora. Dekorator obsługuje funkcje sprawdzające zakresy, typy, wartości i wszystko, co można sobie tylko wymarzyć w tak ekspresyjnym języku jak Python. Ponadto kod został nieznacznie zmieniony w celu usunięcia powtarzających się fragmentów i automatycznego przetwarzania negatywnych wyników testów. Przykłady użycia i oczekiwane wyniki zawarte są w testach do tego modułu. Zgodnie z przedstawionymi wcześniej uwagami do tego przykładu ten dekorator w podanej postaci nie działa w pełni poprawnie w trybie zagnieżdżania — argumenty pozycyjne są weryfikowane jedynie na najgłębszej zagnieżdżonym poziomie. Jednak testy różnych rodzajów można połączyć w jeden test i zastosować w jednym dekoratorze (choć ilość kodu, który trzeba w tym celu wpisać, może zakwestionować korzyści w porównaniu z prostą asercją!).

"""

Plik argtest.py (wersje 3.x i 2.x):

Funkcja dekoratora wykonująca dowolną zadaną weryfikację argumentów innej funkcji lub metody. Dwa przykłady to sprawdzenie zakresu i typu.

Funkcja valuetest wykonuje bardziej ogólne testy wartości argumentów.

Argumenty w dekoratorze są określone za pomocą słów kluczowych.

We właściwym wywołaniu argumenty mogą być przekazywane za pomocą pozycji lub słów kluczowych, a domyślne argumenty mogą być pomijane.

Przykłady użycia pokazane są w samotestującym kodzie niżej.

Uwagi: dekorator nie obsługuje w pełni zagnieżdżania, ponieważ argumenty

funkcji pośredniczącej są inne. Nie można weryfikować dodatkowych argumentów przekazanych do argumentu *args dekorowanej funkcji.

W szczególnych przypadkach może to być bardziej skomplikowany sposób

niż użycie asercji.

"""

```

trace = False

def rangetest(**argchecks):
    return argtest(argchecks, lambda arg, vals: arg < vals[0] or arg
> vals[1])

def typetest(**argchecks):
    return argtest(argchecks, lambda arg, type: not isinstance(arg,
type))

def valuetest(**argchecks):
    return argtest(argchecks, lambda arg, tester: not tester(arg))

def argtest(argchecks, failif):      # Weryfikacja argumentów +
kryteria w failif

    def onDecorator(func):           # onCall zachowuje func,
argchecks, failif

        if not __debug__:           # Brak działań w przypadku
użycia "python -O main.py argumenty..."

            return func

        else:

            code = func.__code__
            expected = list(code.co_varnames[:code.co_argcount])

            def onError(argname, criteria):
                errfmt = 'funkcja %s: argument "%s" nie spełnia
kryterium %s'
                raise TypeError(errfmt % (func.__name__, argname,
criteria))

            def onCall(*pargs, **kargs):
                positionals = expected[:len(pargs)]
                for (argname, criteria) in argchecks.items(): # Dla
wszystkich sprawdzanych
# argumentów

                    if argname in kargs:                      #
przekazanych za pomocą nazwy
                        if failif(kargs[argname], criteria):
                            onError(argname, criteria)

                    elif argname in positionals:                 #
przekazanych za pomocą pozycji
                        position = positionals.index(argname)
                        if failif(pargs[position], criteria):
                            onError(argname, criteria)

    return onCall

```

```

        else:                                #
Argumenty domyślne
            if trace:
                print('Argument "%s" ma domyślną wartość' %
argname)
            return func(*pargs, **kargs)          # OK:
wywołanie oryginalnej funkcji
        return onCall
    return onDecorator
if __name__ == '__main__':
    import sys
    def fails(test):
        try: result = test()
        except: print('[%s]' % sys.exc_info()[1])
        else: print('?%s?' % result)
    print('-----')
    # Przykłady użycia: zakresy, typy
@rangetest(d=(1, 31), m=(1, 12), y=(1900, 2013))
def date(d, m, y):
    print('data: %s-%s-%s' % (d, m, y))
date(2, 1, 1960)
fails(lambda: date(1, 2, 3))
@typetest(a=int, c=float)
def sum(a, b, c, d):
    print(a + b + c + d)
sum(1, 2, 3.0, 4)
sum(1, d=4, b=2, c=3.0)
fails(lambda: sum('spam', 2, 99, 4))
fails(lambda: sum(1, d=4, b=2, c=99))
print('-----')
# Dowolne, różnorodne testy
@valuetest(word1=str.islower, word2=(lambda x: x[0].isupper()))
def msg(word1='mighty', word2='Larch', label='The'):
    print('%s %s %s' % (label, word1, word2))

```

```

msg()                      # Domyślne wartości word1 i word2
msg('majestic', 'Moose')
fails(lambda: msg('Giant', 'Redwood'))
fails(lambda: msg('great', word2='elm'))
print('-----')
# Ręczne sprawdzanie typu i zakresu
@valuetest(A=lambda x: isinstance(x, int), B=lambda x: x > 0 and
x < 10)
def manual(A, B):
    print(A + B)
manual(100, 2)
fails(lambda: manual(1.99, 2))
fails(lambda: manual(100, 20))
print('-----')
# Zagnieżdżanie: uruchomienie obu testów poprzez zagnieżdżenie
# funkcji pośredniczącej w oryginalnej.
# Problem: zewnętrzne poziomy nie sprawdzają argumentów
# pozycyjnych z powodu innej sygnatury funkcji pośredniczącej.
# Gdy trace=True, wszystkie X oprócz ostatniego są klasyfikowane
# jako domyślne z powodu sygnatury funkcji pośredniczącej.
@rangetest(X=(1, 10))
@typetest(Z=str)           # Argumenty pozycyjne
weryfikuje tylko najgłębszy poziom
def nester(X, Y, Z):
    return('%s-%s-%s' % (X, Y, Z))
    print(nester(1, 2, 'spam'))      # Oryginalna funkcja jest
wykonywana poprawnie
fails(lambda: nester(1, 2, 3))      # Wywołana zagnieżdżona
funkcja typetest: argumenty pozycyjne
fails(lambda: nester(1, 2, Z=3))      # Wywołana zagnieżdżona
funkcja typetest: słowa kluczowe
fails(lambda: nester(0, 2, 'spam'))  # <== Zewnętrzna funkcja
rangetest nie jest wykonywana:
# argumenty pozycyjne
fails(lambda: nester(X=0, Y=2, Z='spam')) # Zewnętrzna funkcja
rangetest jest wykonywana: słowa
# kluczowe

```

Wyniki autotestu modułu w wersjach 3.x i 2.x są takie jak niżej (szczegóły niektórych obiektów w wersji 2.x mogą się nieznacznie różnić). Aby uzyskać więcej informacji, jak zwykle należy przeanalizować kod źródłowy.

```
c:\code> py -3 argtest.py
-----
-
data: 2-1-1960
[funkcja date: argument "y" nie spełnia warunku (1900, 2013)]
10.0
10.0
[funkcja sum: argument "a" nie spełnia warunku <class 'int'>]
[funkcja sum: argument "c" nie spełnia warunku <class 'float'>]
-----
-
The mighty Larch
The majestic Moose
[funkcja msg: argument "word1" nie spełnia warunku <method
'islower' of 'str' objects>]
[funkcja msg: argument "word2" nie spełnia warunku <function
<lambda> at 0x01B7F808>]
-----
-
102
[funkcja manual: argument "A" nie spełnia warunku <function
<lambda> at 0x01B7F6A0>]
[funkcja manual: argument "B" nie spełnia warunku <function
<lambda> at 0x01B7F580>]
-----
-
1-2-spam
[funkcja nester: argument "Z" nie spełnia warunku <class 'str'>]
[funkcja nester: argument "Z" nie spełnia warunku <class 'str'>]
?0-2-spam?
[funkcja onCall: argument "X" nie spełnia warunku (1, 10)]
Na koniec, jak już wiemy, tego rodzaju dekorator można stosować zarówno z funkcjami, jak i metodami:
```

```
# Plik argtest_testmeth.py
from argtest import rangetest, typetest
class C:
```

```
@rangetest(a=(1, 10))
def meth1(self, a):
    return a * 1000
@typetest(a=int)
def meth2(self, a):
    return a * 1000
>>> from argtest_testmeth import C
>>> X = C()
>>> X.meth1(5)
5000
>>> X.meth1(20)
TypeError: funkcja meth1: argument "a" nie spełnia kryterium (1,
10)
>>> X.meth2(20)
20000
>>> X.meth2(20.9)
TypeError: funkcja meth2: argument "a" nie spełnia kryterium <class
'int'>
```

Rozdział 40. Metaklasy

W poprzednim rozdziale omówiliśmy dekoratory i zapoznaliśmy się z różnymi przykładami ich użycia. W ostatnim rozdziale książki nadal pozostaniemy w kręgu tworzenia narzędzi i przedstawimy kolejne zaawansowane zagadnienie — *metaklasy* (ang. *metaclasses*).

W pewnym sensie metaklasy po prostu rozszerzają model wstawiania kodu dekoratorów. Jak wiemy z poprzedniego rozdziału, dekoratory funkcji i klas pozwalają nam przechwytywać i rozszerzać wywołania funkcji oraz wywołania tworzące instancje klas. W podobny sposób metaklasy pozwalają nam przechwytywać i rozszerzać *tworzenie klas* — udostępniają API służące do wstawiania dodatkowej logiki, która ma być wykonana na zakończenie instrukcji `class`. Metaklasy robią to jednak w sposób odmienny od generatorów. Tym samym udostępniają ogólny protokół zarządzania obiektami klas w programie.

Tak jak wszystkie kwestie omawiane w tej części książki, jest to *zagadnienie zaawansowane*, z którym można się zapoznać w miarę potrzeby. W praktyce metaklasy pozwalają nam uzyskać wyższy poziom kontroli nad działaniem zbioru klas. Mają one spore możliwości i nie są przeznaczone dla większości programistów aplikacji (ani też, mówiąc szczerze, dla osób o słabym sercu!).

Z drugiej strony metaklasy otwierają drzwi do różnych wzorców programowania, które w inny sposób mogą być trudne lub niemożliwe do osiągnięcia, i są szczególnie interesujące dla programistów chcących pisać elastyczne API czy narzędzia programistyczne przeznaczone dla innych osób. Nawet jeśli ktoś nie mieści się w tej kategorii osób, metaklasy mogą go sporo nauczyć o ogólnym modelu klas Pythona i o wymogach, jakie musi spełnić kod, w którym są wykorzystywane (jak się przekonamy, mają one wpływ nawet na *dziedziczenie*). Metaklasy, tak jak inne zaawansowane narzędzia, zaczęły być stosowane częściej, niż zakładali to ich wynalazcy.

Tak jak w poprzednim rozdziale, również tutaj naszym celem jest zaprezentowanie nieco bardziej realistycznych przykładów kodu, niż miało to miejsce we wcześniejszych częściach książki. Choć metaklasy są zagadnieniem z dziedziny samego jądra języka i nie przynależą do dziedziny aplikacji, celem części niniejszego rozdziału jest wzbudzenie chęci zapoznania się z bardziej rozbudowanymi przykładami programowania aplikacji po zakończeniu książki.

Ponieważ jest to ostatni techniczny rozdział książki, zawiera na początku podsumowanie pewnych zagadnień dotyczących samego Pythona, którymi zajmowaliśmy się do tej pory, a na końcu wnioski. Oczywiście, co robić dalej po przeczytaniu tej książki, jest indywidualną decyzją każdego czytelnika. Jednak w pracy nad otwartym projektem ważne jest, aby mieć w głowie jego całościowy obraz, a jednocześnie zajmować się szczegółami.

Tworzyć metaklasy czy tego nie robić?

Metaklasy są chyba najbardziej zaawansowanym zagadnieniem omawianym w niniejszej książce — jeśli nie w ogóle w Pythonie. Pozwól sobie zapozyczyć cytat z listy dyskusyjnej `comp.lang.python`, napisany przez tworzącego jądro Pythona programistę weterana Tima Petersa (który jest także autorem słynnego motta z `import this`):

[Metaklasy] to magia wyższego poziomu, którą 99% użytkowników Pythona nigdy nie będzie musiało zwracać sobie głowy. Jeśli zastanawiasz się, czy są ci one potrzebne, to nie

są (osoby, które faktycznie będą ich potrzebowaly, będą o tym wiedziały z całą pewnością i nie będą potrzebowaly wyjaśnień, dlaczego tak jest).

Innymi słowy: metaklasy przeznaczone są przede wszystkim dla programistów tworzących API i narzędzi, z których będą korzystały inne osoby. W wielu przypadkach (jeśli nie ich większości) nie będą najlepszym wyborem w pracy nad aplikacjami. Jest tak szczególnie wtedy, gdy tworzy się kod, z którego w przyszłości będą korzystały inne osoby. Pisanie czegoś dlatego, że „wydaje się fajne”, nie jest zazwyczaj rozsądny uzasadnieniem, o ile oczywiście akurat nie eksperymentujemy albo się nie uczymy.

Mimo to metaklasy mają szeroką gamę potencjalnych zastosowań i dobrze jest wiedzieć, kiedy mogą się przydać. Przykładowo można je wykorzystać do ulepszania klas za pomocą opcji takich jak śledzenie, trwałość obiektów czy logowanie wyjątków. Można ich także użyć między innymi do konstruowania części klasy w czasie wykonywania w oparciu o pliki konfiguracyjne, do uniwersalnego stosowania dekoratorów funkcji do każdej metody klasy oraz do sprawdzania zgodności z oczekiwanyimi interfejsami.

W bardziej zaawansowanych wcieleniach metaklasy można nawet wykorzystać do implementowania alternatywnych wzorców kodu, na przykład takich jak programowanie zorientowane aspektowo czy odwzorowania obiektowo-relacyjne (ORM) dla baz danych. Choć często istnieją alternatywne sposoby uzyskiwania takich rezultatów (jak zobaczymy, role dekoratorów klasy oraz metaklas często się pokrywają), metaklasy udostępniają model formalny przygotowany z myślą o tych zadaniach. Nie mamy tutaj miejsca, by omówić w tym rozdziale tego typu zastosowania, jednak po zapoznaniu się z podstawami zachęcam każdego do poszukania w internecie dodatkowych przypadków użycia.

Najbardziej istotnym dla niniejszej książki uzasadnieniem zapoznania się z metaklasami jest to, że zagadnienie to może wspomóc poznanie ogólnej mechaniki klas Pythona. Zobaczmy na przykład, że stanowią one nieodłączną część modelu dziedziczenia klas w nowym stylu, który zostanie tu ostatecznie w pełni sformalizowany. Choć każdy może kiedyś tworzyć lub wykorzystywać utworzone przez kogoś metaklasy, ale nie musi tego robić, zrozumienie podstaw działania metaklas daje głębsze zrozumienie samego Pythona^[1].

Zwiększające się poziomy magii

Większość niniejszej książki skupiała się na prostych technikach tworzenia kodu aplikacji, gdyż na ogólny programiści większość czasu poświęcają na pisanie modułów, funkcji oraz klas służących do rzeczywistych celów. Programiści ci mogą wykorzystywać klasy i tworzyć instancje, a nawet w jakimś stopniu przeciągać operatory, jednak raczej nie będą się zbytnio zagłębiać w to, jak ich klasy naprawdę działają.

W niniejszej książce widzieliśmy także różne narzędzia pozwalające w uniwersalny sposób kontrolować działanie Pythona, które często mają więcej wspólnego z mechanizmami wewnętrznymi Pythona i tworzeniem narzędzi niż z dziedziną programowania aplikacji. Poniżej przedstawione jest przypomnienie, które pomoże nam umiejscowić metaklasy w spektrum narzędzi:

Atrybuty introspektywne

Atrybuty specjalne, takie jak `__class__` czy `__dict__`, pozwalają nam badać wewnętrzne aspekty implementacyjne obiektów Pythona, tak by można je było przetwarzać w sposób ogólny — wyświetlić wszystkie atrybuty obiektu czy nazwę klasy. Dodatkowo, jak widzieliśmy, podobną rolę mogą pełnić narzędzia takie jak metody `dir` i `getattr`, gdy muszą być obsługiwane „wirtualne” atrybuty, np. sloty.

Metody przeciążania operatorów

Metody o specjalnych nazwach, takie jak `__str__` oraz `__add__`, zakodowane w klasach przechwytyują i udostępniają zastosowane do instancji klas wbudowane działania, takie jak

wyświetlanie czy operatory wyrażeń. Są one wykonywane automatycznie w odpowiedzi na wbudowane operacje i pozwalają klasom na zachowanie zgodności z oczekiwanyimi interfejsami.

Metody przechwytywania atrybutów

Specjalna kategoria metod przeciążania operatorów udostępnia sposób przechwytywania dostępu do atrybutów w sposób ogólny — `__getattr__`, `__setattr__`, `__delattr__` oraz `__getattribute__` pozwalając klasom opakowującym na wstawianie automatycznie wykonywanego kodu, który może sprawdzać poprawność żądań atrybutów i delegować je do obiektów osadzonych. Pozwalają one na obliczenie przy dostępie dowolnej liczby atrybutów obiektu — albo wybranych atrybutów, albo wszystkich w ogóle.

Właściwości klas

Wbudowana funkcja `property` pozwala na wiązanie z określonym atrybutem klasy kodu, który jest wykonywany automatycznie przy pobieraniu, przypisywaniu lub usuwaniu atrybutu. Choć właściwości nie są tak uniwersalne jak narzędzia z poprzedniego akapitu, pozwalają one na automatyczne wywoływanie kodu przy dostępie do określonych atrybutów.

Deskryptory atrybutów klas

Tak naprawdę funkcja `property` jest zwięzłym sposobem definiowania deskryptora atrybutu, który automatycznie wykonuje funkcje w momencie dostępu. Deskryptory pozwalają definiować odrębne metody obsługi klas `__get__`, `__set__` oraz `__delete__`, wykonywane automatycznie w momencie dostępu do atrybutu przypisanego do instancji tej klasy. Udostępniają one ogólny sposób wstawiania automatycznie wykonywanego kodu w momencie dostępu do określonego atrybutu i są uruchamiane po normalnym wyszukaniu atrybutu.

Dekoratory funkcji i klas

Jak widzieliśmy w rozdziale 39., specjalna składnia dekoratorów z `@` pozwala nam dodawać logikę, która zostanie wykonana automatycznie po wywołaniu funkcji lub utworzeniu instancji klasy. Ta logika opakowująca może śledzić lub ustawać w czasie wywołania, sprawdzać poprawność argumentów, zarządzać wszystkimi instancjami klasy, rozszerzać instancje o dodatkowe działania, takie jak sprawdzanie poprawności pobieranych atrybutów i wiele innych. Składnia dekoratorów wstawia logikę dowiązywania nazw, która ma być wykonywana na końcu instrukcji definicji funkcji oraz klas — nazwy udekorowanych funkcji i klas są ponownie dowiązywane do obiektów, które można wywoływać, przechwytyujących późniejsze wywołania.

Metaklasy

Ostatnie magiczne narzędzia, wspomniane w rozdziale 32., którymi się tutaj zajmiemy.

Jak wspomniano we wprowadzeniu do niniejszego rozdziału, *metaklasy* są kontynuacją tej tematyki — pozwalają wstawiać logikę, która będzie wykonywana automatycznie, kiedy tworzony jest obiekt klasy — na końcu instrukcji `class`. Logika ta nie dowiązuje ponownie nazwy klasy do obiektu wywoływalnego dekoratora, a raczej przekierowuje *samo tworzenie klasy* do wyspecjalizowanej logiki.

Język pełen haczyków

Innymi słowy, metaklasy są właściwie innym sposobem definiowania *kodu wykonywanego automatycznie*. Za pomocą metaklas oraz pozostałych wymienionych narzędzi Python umożliwia nam wstawianie logiki w różnych kontekstach — w momencie analizy operatorów, dostępu do atrybutów, wywołań funkcji, tworzenia instancji klas, a teraz tworzenia obiektów klas. Jest to język z mnóstwem *punktów zaczepienia* — funkcjonalności otwartej na nadużycia jak każda

inna, ale też oferującej elastyczność, której niektórzy programiści pożądają, a w niektórych programach jest ona niezbędna.

Jak widzieliśmy, wiele z tych zaawansowanych narzędzi Pythona ma role, które się na siebie nakładają. Atrybutami można na przykład zarządzać za pomocą właściwości, deskryptorów lub metod przechwytywania atrybutów. Jak zobaczymy w niniejszym rozdziale, dekoratorów klas oraz metaklas można także używać zamiennie. Dla przypomnienia:

- Choć *dekoratory klas* są często wykorzystywane do zarządzania instancjami, można ich zamiast tego użyć do zarządzania klasami.
- I podobnie — choć *metaklasy* zaprojektowano w celu rozszerzania konstrukcji klas, mogą także wstawiać kod, zarządzając tym samym instancjami.

W rzeczywistości główna różnica pomiędzy obydwooma narzędziami polega na *chwili*, w której są stosowane w czasie tworzenia klasy. Jak wiemy z poprzedniego rozdziału, dekorator klasy jest uruchamiany po utworzeniu klasy. Z tego powodu dekoratory są stosowane do definiowania kodu uruchamianego w chwili tworzenia instancji. Jeżeli faktycznie wprowadzają one dodatkowy kod do klasy, zazwyczaj odbywa się to z użyciem obiektu pośredniczącego, a nie przez bardziej bezpośrednią relację.

Natomiast metaklasy, jak zobaczymy w tym rozdziale, są uruchamiane w *chwili* tworzenia klasy. W efekcie kodowi klienckiemu są zwracane nowe klasy. Dlatego metaklasy są często wykorzystywane do zarządzania i modyfikowania samych *klas*. Mogą nawet zawierać metody do przetwarzania za pomocą bezpośrednich relacji utworzonych przez nie klas.

Przykładowo metaklasy można wykorzystać między innymi do automatycznego dodania dekoracji do wszystkich metod klas, rejestrowania wszystkich klas do użycia w API, automatycznego dodawania logiki interfejsu użytkownika czy tworzenia lub rozszerzania klas z uproszczonych specyfikacji w plikach tekstowych. Ponieważ możemy kontrolować sposób tworzenia klas (a także za pomocą pośrednika zachowanie ich instancji), dziedzina zastosowania jest w tym przypadku bardzo szeroka.

Jak się przekonamy w tym rozdziale, pomiędzy obydwooma narzędziami pod wieloma względami istnieje więcej podobieństw niż różnic. Ponieważ wybór, z której techniki korzystać, jest czasami bardzo subiektywny, znajomość alternatywnych rozwiązań może pomóc wybrać narzędzie odpowiednie dla zadania. Aby lepiej zrozumieć dostępne opcje, sprawdźmy, jak działają metaklasy.

Wady funkcji pomocniczych

Tak jak dekoratory z poprzedniego rozdziału, metaklasy teoretycznie często są opcjonalne. Zazwyczaj ten sam efekt możemy uzyskać, przekazując obiekty klas za pomocą *funkcji zarządzających* (nazywanych czasami także funkcjami pomocniczymi), w podobny sposób, jak możemy osiągnąć cele uzyskiwane za pomocą dekoratorów, przekazując funkcje i instancje za pomocą kodu zarządzającego. Podobnie jak dekoratory, tak i metaklasy:

- Udostępniają bardziej formalną i jawną strukturę.
- Pomagają upewnić się, że programiści aplikacji nie zapomną rozszerzyć klas zgodnie z wymaganiami API.
- Pozwalają uniknąć powtarzalności kodu i związanych z nią kosztów utrzymywania, dokonując faktoryzacji logiki dostosowującej klasę do własnych potrzeb w jednym miejscu — w metaklasie.

By to zilustrować, założymy, że chcemy automatycznie wstawić metodę do zbioru klas. Oczywiście moglibyśmy to zrobić za pomocą prostego *dziedziczenia*, jeśli metoda jest znana w momencie tworzenia kodu klas. W takim przypadku kod metody wpisujemy do klasy nadrzędej, a wszystkie klasy docelowe będą ją stamtąd dziedziczyć:

```

class Extras:

    def extra(self, args):                      # Normalne dziedziczenie:
        zbyt statyczne

    ...

class Client1(Extras): ...                    # Klasy Client dziedziczą
dodatkowe metody

class Client2(Extras): ...

class Client3(Extras): ...

X = Client1()                                # Utworzenie instancji

X.extra()                                     # Wykonanie dodatkowych
metod

```

Czasami jednak nie da się przewidzieć takiego rozszerzenia na etapie pisania kodu klas. Rozważmy przypadek, w którym klasy rozszerzane są w odpowiedzi na wybór dokonany przez użytkownika w czasie wykonywania lub specyfikację wpisaną do pliku konfiguracyjnego. Choć moglibyśmy napisać kod każdej klasy w naszym wyimaginowanym zbiorze, tak by to *ręcznie* sprawdzał, jest to naprawdę spore wymaganie w stosunku do klientów (`required` jest tu abstrakcyjne — to coś, co ma być wypełnione):

```

def extra(self, arg): ...

class Client1: ...                            # Klasa Client rozszerza:
zbyt rozdrobnione

if required():
    Client1.extra = extra

class Client2: ...

if required():
    Client2.extra = extra

class Client3: ...

if required():
    Client3.extra = extra

X = Client1()

X.extra()

```

Możemy w ten sposób dodawać metody do klasy już po instrukcji `class`, ponieważ metoda klasy jest po prostu funkcją powiązaną z klasą i ma pierwszy argument, w którym przekazujemy instancję `self`. Choć powyższe rozwiązywanie działa, cały ciężar rozszerzania kładzie się na klasach klienta (i zakłada, że to one mają pamiętać, by to wszystko zrobić!).

Z punktu widzenia utrzymywania kodu lepszym rozwiązyaniem byłoby wyizolowanie logiki wyboru w jednym miejscu. Możemy hermetyzować część tych dodatkowych działań, przekierowując klasy za pomocą *funkcji zarządzającej*. Taka funkcja zarządzająca rozszerzyłaby klasę zgodnie z wymaganiami i poradziła sobie z wszystkimi kwestiami wynikającymi z testowania w czasie wykonania oraz konfiguracją:

```
def extra(self, arg): ...
```

```

def extras(Class):                                # Funkcja zarządzająca:
    zbyt ręczna

    if required():
        Class.extra = extra

class Client1: ...

extras(Client1)

class Client2: ...

extras(Client2)

class Client3: ...

extras(Client3)

X = Client1()

X.extra()

```

Powyższy kod powoduje wykonanie klasy za pośrednictwem funkcji zarządzającej bezpośrednio po utworzeniu. Choć funkcje zarządzające tego typu mogą tutaj spełnić nasze cele, nadal zbyt duży ciężar spoczywa na twórcach klas, którzy muszą rozumieć wymagania i stosować się do nich w kodzie. Byłoby lepiej, gdyby istniał prosty sposób wymuszania rozszerzania w klasach docelowych, tak by musiały się one zajmować rozszerzaniem ani o nim nie zapominały. Innymi słowy, chciałibyśmy móc wstawić na końcu instrukcji `class` jakiś kod, który będzie wykonywany *automatycznie* i spowoduje rozszerzenie klasy.

Właśnie do tego służą *metaklasy*. Deklarując metaklasę, mówimy Pythonowi, by przekierował tworzenie obiektu klasy do innej przekazanej mu klasy:

```

def extra(self, arg): ...

class Extras(type):

    def __init__(Class, classname, superclasses, attributedict):
        if required():
            Class.extra = extra

class Client1(metaclass=Extras): ...           # Jedynie deklaracja
metaklasy

class Client2(metaclass=Extras): ...           # Klasa Client jest
instancją metaklasy

class Client3(metaclass=Extras): ...

X = Client1()                                  # X jest instancją klasy
Client1

X.extra()

```

Ponieważ Python wywołuje metaklasę automatycznie na końcu instrukcji `class`, kiedy tworzona jest nowa klasa, może rozszerzać klasę, rejestrować ją lub w inny sposób nią zarządzać. Co więcej, jedynym wymaganiem ze strony klas klienta jest to, że muszą one deklarować metaklasę. Każda klasa, która to robi, automatycznie pobiera rozszerzenie udostępniane przez metaklasę — zarówno teraz, jak i w przyszłości, jeśli metaklasa się zmieni.

Oczywiście jest to typowy przykład, którego przydatność należy ocenić samodzielnie. W rzeczywistości w kodzie klienckim można równie łatwo zapomnieć o zastosowaniu metaklasy,

podobnie jak o wywołaniu funkcji zarządzającej! Niemniej jednak w przypadku metaklas, dzięki jej jawnej naturze, jest to mniej prawdopodobne. Co więcej, metaklasy oferują dodatkowe możliwości, których jeszcze nie poznaliśmy. Choć może być trudno zobaczyć to w powyższym krótkim przykładzie, metaklasy zazwyczaj radzą sobie z takimi zadaniami lepiej od pozostałych rozwiązań.

Metaklasy a dekoratory klas — runda 1.

Warto również zauważyć, że opisane w poprzednim rozdziale *dekoratory klas* czasami nakładają się z metaklasami, zarówno pod względem działania, jak i przydatności. Choć zazwyczaj wykorzystywane są do zarządzania instancjami lub rozszerzania ich, dekoratory klas mogą także rozszerzać klasy niezależnie od wszelkich utworzonych instancji. Dzięki ich składni metaklasy stosuje się w bardziej jawnym i zdecydowanie bardziej oczywistym sposobie niż funkcje zarządzające.

Przypuśćmy na przykład, że nasza funkcja zarządzająca ma zwracać rozszerzoną klasę, zamiast po prostu modyfikować ją w miejscu. Pozwoli to na większą elastyczność, ponieważ funkcja zarządzająca może zwracać dowolny typ obiektu implementujący oczekiwany interfejs klasy:

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

class Client1: ...
Client1 = extras(Client1)

class Client2: ...
Client2 = extras(Client2)

class Client3: ...
Client3 = extras(Client3)

X = Client1()
X.extra()
```

Osoby, którym się wydaje, że kod ten zaczyna przypominać dekoratory klas, mają rację. W poprzednim rozdziale zaprezentowaliśmy dekoratory klas jako narzędzie służące do rozszerzania wywołań tworzących *instancje*. Ponieważ jednak działają one automatycznie, dowiązując nazwę klasy do wyniku funkcji, nie ma żadnego powodu, byśmy nie mogli użyć ich do rozszerzenia klasy przed utworzeniem jakichkolwiek instancji. Oznacza to, że dekoratory klas mogą zastosować dodatkową logikę do *klas*, a nie tylko *instancji*, w momencie tworzenia:

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

@extras
```

```

class Client1: ...                                # Client1 = extras(Client1)

@extras

class Client2: ...                                # Ponownie dowiązuje klasę
niezależnie od instancji

@extras

class Client3: ...

X = Client1()                                     # Tworzyinstancję rozszerzonej
klasy

X.extra()                                         # X jest instancją oryginalnej
klasy Client1

```

Dekoratory zasadniczo automatyzują tutaj ręczne dowiązywanie nazwy z poprzedniego przykładu. Tak jak w przypadku metaklas, ponieważ dekorator zwraca oryginalną klasę, instancje są tworzone z niego, a nie z obiektu opakowującego. Tak naprawdę tworzenie instancji w ogóle nie jest przechwytywane.

W tym akurat przypadku — dodawania metod do klasy, kiedy jest ona tworzona — wybór pomiędzy metaklasami a dekoratorami jest dość dowolny. Dekoratory można wykorzystać do zarządzania *zarówno* instancjami, jak i klasami, a w tej drugiej roli pokrywają się one z metaklasami. Jednak wybór nie jest taki jednoznaczny. W rzeczywistości role obu narzędzi są zdeterminowane ich sposobem działania.

Jak zobaczymy, dekoratory są w tej roli odpowiednikiem metod `__init__` metaklasy. Jednak te metaklasy mają dodatkowe punkty zaczepienia pozwalające na dostosowanie do własnych potrzeb. Jak się również przekonamy, poza inicjalizacją klas, metaklasy mogą wykonywać dowolne zadania konstrukcyjne, które mogą być trudniejsze do wykonania za pomocą dekoratorów. Z tego powodu są nieco bardziej skomplikowane, ale za to lepiej przystosowane do modyfikowania klas w chwili ich tworzenia.

Na przykład metaklasa również ma metodę `__new__` wykorzystywaną podczas tworzenia klasy, jednak nie ma tu analogii do dekoratora. Tworzenie nowej klasy za pomocą dekoratora wymaga wykonania dodatkowego kroku. Co więcej, za pomocą metaklasy można do klas wprowadzać funkcjonalności podobne do metod, czego również nie można osiągnąć za pomocą dekoratorów (przy ich użyciu trzeba to robić w mniejszej bezpośredni sposób).

Co więcej, metaklasy zaprojektowano do zarządzania klasami i zastosowanie ich dla celów zarządzania *instancjami* jest o wiele mniej proste. Ponieważ są one odpowiedzialne również za tworzenie samych klas, wprowadzają dodatkowy krok do roli zarządzania instancjami.

Omówimy te różnice na przykładzie kodu nieco później w niniejszym rozdziale, a fragmentarny kod zastąpimy prawdziwym, działającym przykładem. By jednak w pełni zrozumieć, jak działają metaklasy, musimy najpierw uzyskać jaśniejszy obraz modelu leżącego u ich podstaw.

Tu jest magia, a wszędzie indziej też jest magia

Lista w podrozdziale „Zwiększające się poziomy magii” dotyczy rodzajów magii innych niż ta, która jest przez programistów powszechnie uznawana za przydatną. Niektórzy mogą do tej listy dodać narzędzia do *programowania funkcyjnego*, na przykład domknięcia i generatory, a nawet podstawowe techniki *programowania obiektowego*. Pierwszy rodzaj programowania opiera się na zachowaniu zakresu i automatycznym tworzeniu obiektów generatora, drugi na przeszukiwaniu atrybutów dziedziczenia i użyciu pierwszego, specjalnego argumentu funkcji. Te dwa podejścia, choć również magiczne, ułatwiają programowanie poprzez wprowadzenie warstwy abstrakcji powyżej i poniżej architektury wykorzystywanej sprzęt.

Programowanie obiektowe jest szeroko stosowane w informatycznym świecie. Oferuje ono model pisania programów, który jest pełniejszy, ścisłej i bardziej strukturalny niż dostępny za pomocą narzędzi funkcyjnych. Oznacza to, że niektóre poziomy magii są uważane za bardziej uzasadnione niż inne. W końcu, gdyby nie magia, programy nadal składałyby się z kodu maszynowego (lub fizycznych przełączników).

Tym, co naraża systemy na ryzyko przekroczenia progu złożoności, jest *nagromadzenie nowej magii*. Jest to na przykład wprowadzenie paradygmatu funkcyjnego do języka, który od zawsze był obiektowy, albo wprowadzenie równoważnych i zaawansowanych sposobów osiągania celów, które przez większość programistów są rzadko stosowane. Taka magia może stanowić poprzeczkę zbyt wysoką do pokonania dla dużej części użytkowników tworzonych narzędzi.

Co więcej, czasami magia może dotyczyć jedną część programistów bardziej niż inną. Na przykład użytkownicy kompilatora tłumaczącego kod nie muszą być twórcami kompilatorów. Natomiast w przypadku funkcji super przyjęto założenie pełnego opanowania i umiejętności stosowania bezsprzecznie niejasnego i sztucznego algorytmu MRO. Opisany w tym rozdziale algorytm *dziedziczenia* w nowym stylu również zakłada znajomość deskryptorów, metaklas i algorytmu MRO, które same w sobie są zaawansowanymi narzędziami. Nawet niejawne „haczyki”, takie jak deskryptory, pozostają niejawne tylko do pierwszej awarii lub cyklu utrzymywianego. Tego rodzaju magia rozszerza zakres wstępnych wymagań i obniża przydatność narzędzia.

W otwartych systemach tylko czas i pobrania mogą określić, gdzie leżą takie przeszkody. Znalezienie równowagi pomiędzy siłą a złożonością narzędzia zależy w również mierze od opinii, jak i technologii. Pomijając subiektywne czynniki, nowa magia narzucona użytkownikom niewątpliwie wypacza krzywą uczenia się systemu. Do tego tematu powrócimy w końcowych słowach następnego rozdziału.

Model metaklasy

By naprawdę zrozumieć, jak działają metaklasy, musimy dowiedzieć się nieco więcej o modelu typów Pythona, a także tym, co dzieje się na końcu instrukcji `class`. Jak się przekonamy, te dwa zagadnienia są ze sobą ściśle związane.

Klasy są instancjami obiektu `type`

Dotychczas w książce wykonywaliśmy większość pracy, tworząc instancje typów wbudowanych, takich jak listy i łańcuchy znaków, a także instancje klas tworzonych przez nas samych. Jak widzieliśmy, instancje *klas* mają pewne własne atrybuty z informacjami o stanie, jednak dziedziczą także atrybuty związane z działaniem po klasach, z których zostały utworzone. Tak samo jest w przypadku typów *wbudowanych* — instancje list mają na przykład własne wartości, jednak dziedziczą metody po typie listy.

Choć z takimi obiektami instancji możemy sporo zrobić, model typów Pythona okazuje się nieco bogatszy, niż to przedstawiłem. Tak naprawdę w modelu opisywanym dotychczas znajduje się pewna luka — skoro instancje tworzone są z klas, co tworzy same klasy? Okazuje się, że także klasy są instancją czegoś:

- W *Pythonie 3.x* obiekty klas zdefiniowanych przez użytkownika są instancjami obiektu o nazwie `type`, który sam jest klasą.
- W *Pythonie 2.x* klasy w nowym stylu dziedziczą po `object` będącym klasą podzieloną `type`. Klasy klasyczne są instancjami `type` i nie są tworzone z klasą.

Omawialiśmy pojęcie typów w rozdziale 9., a związek klas z typami w rozdziale 32., jednak wróćmy tutaj do podstaw, by pokazać, jakie zastosowanie ma to do metaklas.

Przypomnijmy, że wbudowana funkcja `type`, wywołana z jednym argumentem, zwraca typ dowolnego obiektu (który sam jest obiektem). W przypadku typów wbudowanych, takich jak listy, typem instancji jest wbudowany typ listy, jednak typ typu listy sam jest typem `type`. Obiekt `type` znajdujący się na górze hierarchii tworzy określone typy, natomiast konkretne typy tworzą instancje. Można to zobaczyć samemu w sesji interaktywnej. Na przykład w Pythonie 3.x typem instancji listy jest klasa `list`, a typem klasy `list` jest klasa `type`:

```
C:\code> py -3                                     # W wersji 3.x:  
>>> type([]), type(type([]))                  # Instancja listy jest  
tworzona na bazie klasy list  
  
(<class 'list'>, <class 'type'>)            # Klasa list jest tworzona na  
bazie klasy type  
  
>>> type(list), type(type)                   # To samo, tylko z nazwami  
typów  
  
(<class 'type'>, <class 'type'>)            # Typem type jest type: szczyt  
hierarchii
```

Zgodnie z tym, czego dowiedzieliśmy się o zmianach dotyczących klas w nowym stylu w rozdziale 32., tak samo jest w Pythonie 2.x, jednak typy nie są do końca tym co klasy — `type` jest unikalnym rodzajem wbudowanego obiektu stojącego na szczytach hierarchii typów i wykorzystywanego do konstruowania typów:

```
C:\code> py -2                                     # W 2.x type jest czymś nieco  
innym  
>>> type([]), type(type([]))                  # W 2.x type jest czymś nieco  
innym  
(<type 'list'>, <type 'type'>)  
>>> type(list), type(type)  
(<type 'type'>, <type 'type'>)
```

Okazuje się, że nasz relacja między typami a instancjami jest prawdziwa również dla klas zdefiniowanych przez użytkownika — instancje tworzone są z klas, a klasy tworzone są z `type`. W Pythonie 3.x pojęcie typu zostało jednak połączone z pojęciem klasy. Tak naprawdę te dwa pojęcia są właściwie synonimami — *klasy są typami, a typy są klasami*. A zatem:

- Typy definiowane są przez klasy pochodzące od `type`.
- Klasy zdefiniowane przez użytkownika są instancjami klas typów.
- Klasy zdefiniowane przez użytkownika są typami generującymi własne instancje.

Jak widzieliśmy wcześniej, ta równoważność wpływa na kod sprawdzający typ instancji — typem instancji jest klasa, z której została ona utworzona. Ma to także wpływ na sposób tworzenia klas, który okazuje się kwestią kluczową dla tematu niniejszego rozdziału. Ponieważ klasy normalnie domyślnie tworzone są z głównej klasy typu, większość programistów nie musi się zastanawiać nad równoważnością typów i klas. Otwiera to jednak przed nami nowe możliwości dostosowywania klas i ich instancji do własnych potrzeb.

Przykładowo klasy w wersji 3.x (a także klasy w nowym stylu z Pythona 2.x) są instancjami klasy `type`, natomiast obiekty instancji są instancjami ich klas. Klasy mają teraz atrybut specjalny `__class__`, który wskazuje `type`, tak samo jak instancje mają atrybut `__class__` wskazujący klasę, z której powstała instancja:

```
C:\code> py -3
```

```

>>> class C: pass          # Obiekt klasy z 3.x (nowy styl)
>>> X = C()                # Obiekt instancji klasy
>>> type(X)               # Instancja jest instancją klasy
<class '__main__.C'>
>>> X.__class__            # Klasa instancji
<class '__main__.C'>
>>> type(C)                # Klasa jest instancją type
<class 'type'>
>>> C.__class__             # Klasa klasy to type
<class 'type'>

```

Warto zwrócić szczególną uwagę na dwa ostatnie wiersze — klasy są instancjami klasy `type` w taki sam sposób, jak normalne instancje są instancjami klasy. Działa to tak samo zarówno dla klas wbudowanych, jak i typów klas zdefiniowanych przez użytkownika w wersji 3.x. Klasy nie są wcale tak naprawdę odrębną koncepcją — to po prostu typy zdefiniowane przez użytkownika, a sam `type` definiowany jest za pomocą klasy.

W Pythonie 2.x działa to podobnie w przypadku klas w nowym stylu, pochodzących od `object`, ponieważ uruchamia to działanie z wersji 3.x (jak widzieliśmy, w wersji 3.x na najwyższym poziomie hierarchii klas automatycznie jest dodawana klasa `object` do krotki `__bases__` superklasy; w ten sposób klasy są kwalifikowane jako utworzone w nowym stylu):

```

C:\code> py -2
>>> class C(object): pass      # W klasach w nowym stylu z 2.x
>>> X = C()                  # także klasy mają klasę
>>> type(X)
<class '__main__.C'>
>>> X.__class__
<class '__main__.C'>
>>> type(C)
<type 'type'>
>>> C.__class__
<type 'type'>

```

Tradycyjne klasy z wersji 2.x są jednak nieco inne — ponieważ odzwierciedlają model klas ze starszych wydań Pythona, nie mają odnośnika `__class__` i tak jak typy wbudowane z wersji 2.x są one instancjami `type`, a nie klasy typu (dla przejrzystości w tym rozdziale skróciłem niektóre adresy szesnastkowe wyświetlanego obiektów):

```

C:\code> py -2
>>> class C: pass          # W tradycyjnych klasach z 2.x
>>> X = C()                # klasy same nie mają klasy
>>> type(X)

```

```
<type 'instance'>
>>> X.__class__
<class __main__.C at 0x005F85A0>
>>> type(C)
<type 'classobj'>
>>> C.__class__
AttributeError: class C has no attribute '__class__'
```

Metaklasy są klasami podrzędnymi klasy type

Dlaczego zatem ma dla nas znaczenie, że w wersji 3.x klasy są instancjami klasy `type`? Okazuje się, że to właśnie jest punkt zaczepienia, który pozwala nam pisać metaklasy. Ponieważ pojęcie *typu* jest obecnie równoznaczne z *klasą*, możemy tworzyć klasy podrzędne dla `type` za pomocą normalnych technik zorientowanych obiektowo, a także składni klas umożliwiającej dostosowanie ich do własnych potrzeb. A ponieważ klasy są tak naprawdęinstancjami klasy `type`, tworzenie ich z dostosowanych do naszych potrzeb klas podrzędnych `type` pozwala na implementację własnych rodzajów klas. Uwzględniając wszystkie szczegóły, całość działa dość naturalnie — w wersji 3.x, a także klasach w nowym stylu z wersji 2.x:

- `type` jest klasą generującą klasy zdefiniowane przez użytkownika.
- Metaklasy są klasami podrzędnymi klasy `type`.
- Obiekty klas są instancjami klasy `type` lub inaczej jej klasami podrzędnymi.
- Obiekty instancji generowane są z klas.

Innymi słowy, by kontrolować sposób tworzenia klas i rozszerzania ich działania, wystarczy podać, że klasa zdefiniowana przez użytkownika ma być tworzona z metaklasą zdefiniowaną przez użytkownika, a nie z normalnej klasy `type`.

Warto zauważyć, że ten związek *instancji typów* nie jest tym samym, co *dziedziczenie*. Klasę zdefiniowaną przez użytkownika także mogą mieć klasy nadrzędne, po których one same oraz ich instancje dziedziczą atrybuty. Jak widzieliśmy, klasy nadrzędne dziedziczenia wymienione są w nawiasach w instrukcji `class` i pokazują się w krotce `__bases__` klasy. Typ, na bazie którego tworzona jest klasa, i którego jest ona instancją, to inny rodzaj związku. W procesie dziedziczenia przeszukiwane są słowniki instancji i przestrzeni nazw klas, jednak klasy mogą przejmować kod ze swoich typów, które nie są ujawniane podczas zwykłego przeszukiwania hierarchii dziedziczenia.

Aby przygotować podstawy do opisu tej różnicy, w kolejnym podrozdziale opisano procedurę stosowaną przez Pythona do implementacji związku „*bycia instancją typu*”.

Protokół instrukcji `class`

Utworzenie klasy podrzędnej dla `type` w celu dostosowania jej do własnych potrzeb to tak naprawdę połowa magii stojącej za metaklasami. Nadal musimy w jakiś sposób skierować tworzenie klasy do metaklasy zamiast do domyślnej `type`. By w pełni zrozumieć sposób wykonywania tego, musimy także wiedzieć, w jaki sposób działają instrukcje `class`.

Wiemy już, że kiedy Python znajduje instrukcję `class`, wykonuje zagnieżdżony w niej blok kodu w celu utworzenia atrybutów klasy — wszystkie nazwy przypisane na najwyższym poziomie zagnieżdzonego bloku kodu generują atrybuty w wynikowym obiekcie klasy. Te nazwy są zazwyczaj funkcjami metod utworzonymi za pomocą zagnieżdżonych instrukcji `def`, jednak

mogą to być także dowolne inne atrybuty przypisane w celu utworzenia danych klasy współdzielonych przez wszystkie jej instancje.

Z technicznego punktu widzenia Python postępuje w tym celu zgodnie ze standardowym protokołem. Na końcu instrukcji `class` i po wykonaniu całego zagnieźdzonego w niej kodu w słowniku przestrzeni nazw wywołuje obiekt `type` w celu utworzenia obiektu klasy:

```
class = type(nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)
```

Obiekt `type` definiuje z kolei metodę przeciążania operatorów `__call__`, która wykonuje dwie pozostałe metody, gdy wywoływany jest obiekt `type`:

```
type.__new__(klaśa_tytu, nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)  
type.__init__(klaśa, nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)
```

Metoda `__new__` tworzy i zwraca nowy obiekt `class`, a następnie metoda `__init__` inicjalizuje nowo utworzony obiekt. Jak zobaczymy za moment, są to właśnie punkty zaczepienia wykorzystywane przez metaklasy (będące klasami podlegającymi `type`) do dostosowywania klas do własnych potrzeb.

Mając na przykład poniższą definicję klasy:

```
class Eggs: ... # Nazwa dziedziczonej klasy  
class Spam(Eggs): # Dziedziczy po Eggs  
    data = 1 # Atrybut danych klasy  
    def meth(self, arg): # Atrybut metody klasy  
        return self.data + arg
```

wewnętrznie Python wykona zagnieźdzony blok kodu w celu utworzenia dwóch atrybutów klasy (`data` oraz `meth`), a następnie wywoła obiekt `type`, by wygenerować obiekt klasy na końcu instrukcji `class`:

```
Spam = type('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

W rzeczywistości, wywołując w ten sposób funkcję `type`, można samodzielnie utworzyć klasę — tutaj z przygotowaną metodą i pustą krotką klasy nadrzędnej (klasa `object` jest dodawana automatycznie w wersjach 3.x i 2.x):

```
>>> x = type('Spam', (), {'data': 1, 'meth': (lambda x, y: x.data + y)})  
>>> i = x()  
>>> x, i  
(<class '__main__.Spam'>, <__main__.Spam object at 0x029E7780>)  
>>> i.data, i.meth(2)  
(1, 3)
```

Utworzona klasa wygląda dokładnie tak samo jak uzyskana za pomocą instrukcji `class`:

```
>>> x.__bases__  
(<class 'object'>,)  
>>> [(a, v) for (a, v) in x.__dict__.items() if not a.startswith('__')]  
[('data', 1), ('meth', <function <lambda> at 0x0297A158>)]
```

Ponieważ wywołanie funkcji `type` odbywa się automatycznie na końcu instrukcji `class`, jest to idealny punkt zaczepienia dla rozszerzania klasy lub przetwarzania jej w inny sposób. Sztuczka polega na zastąpieniu `type` za pomocą własnej klasy podrzędnej, która przechwyci to wywołanie. W kolejnym podrozdziale pokażemy, jak to zrobić.

Deklarowanie metaklas

Jak widzieliśmy przed chwilą, klasy domyślnie tworzone są za pomocą klasy `type`. Żeby przekazać Pythonowi, by tworzył klasę za pomocą własnej metaklasy, musimy po prostu zadeklarować metaklasę w taki sposób, by przechwytywała ona normalne wywołania tworzące klasy. Sposób wykonania tego uzależniony jest od wykorzystywanej wersji Pythona.

Deklarowanie w wersji 3.x

W Pythonie 3.x należy wymienić pożądaną metaklasę jako argument ze słowem kluczowym w nagłówku instrukcji `class`:

```
class Spam(metaclass=Meta):           # Tylko wersja 3.x
```

Klasy nadrzędne, po których się dziedziczy, także można wymienić w nagłówku. W poniższym przykładzie nowa klasa `Spam` dziedziczy po `Eggs`, ale jest równocześnie instancją klasy `Meta`, przez którą jest tworzona:

```
class Spam(Eggs, metaclass=Meta):      # Dopuszczalne inne klasy nadrzędne
```

W powyższym zapisie klasy nadrzędne muszą znajdować się przed metaklasami, ponieważ mają tu zastosowanie zasady kolejności definiowania nazwanych argumentów funkcji.

Deklarowanie w wersji 2.x

Ten sam efekt uzyskamy w Pythonie 2.x, jednak metaklasę musimy podać w inny sposób — za pomocą *atrybutu klasy*, a nie argumentu ze słowem kluczowym.

```
class Spam(object):                  # Tylko wersja 2.x, object
opcjonalny

    __metaclass__ = Meta

class Spam(Eggs, object):           # Zwykła klasa nadrzędna OK, object
opcjonalny

    __metaclass__ = Meta
```

Z technicznego punktu widzenia, aby niektóre klasy w wersji 2.x mogły korzystać z metaklas, *nie* muszą dziedziczyć klasy `object`. Uniwersalny mechanizm kierowania metaklasami został wprowadzony w tym samym czasie, co klasy w nowym stylu, jednak obie funkcjonalności nie są ze sobą powiązane. Jednak deklaracja `__metaclass__` tworzy metaklasę. W wersji 2.x wynikowa klasa w nowym stylu jest tworzona automatycznie poprzez dodanie klasy `object` do sekwencji `__bases__`. Jeżeli powyższej deklaracji nie ma, wtedy w wersji 2.x jest domyślnie wykorzystywany zwykły mechanizm tworzenia klas. Z tego powodu niektóre klasy w wersji 2.x wymagają użycia jedynie atrybutu `__metaclass__`.

Z drugiej strony należy zwrócić uwagę, że użycie metaklasy w wersji 2.x powoduje utworzenie klasy w nowym stylu nawet bez *jawnego* odwołania do klasy `object`. Taka klasa funkcjonuje w nieco innym sposobie niż opisany w rozdziale 32. Jak się za chwilę przekonamy, w wersji 2.x

tworzona klasa lub jej klasa nadrzędna musi jawnie dziedziczyć klasę `object`, ponieważ klasa w nowym stylu nie może w tym kontekście dziedziczyć wyłącznie typowej klasy nadrzędnej. Zatem dziedziczenie klasy `object` nie jest sprzeczne z naturą klasy i może być niezbędne do uniknięcia potencjalnych problemów.

W Pythonie 2.x zmienna globalna `modułu __metaclass__` jest dostępna i łączy wszystkie klasy w module z metaklasą. W wersji 3.x nie jest to już obsługiwane, gdyż zamierzone było jako tymczasowy sposób ułatwiający domyślne tworzenie klas w nowym stylu bez wyprowadzania każdej klasy od `object`. Ponadto w wersji 3.x ignorowane są atrybuty klasy z wersji 2.x, natomiast składnia z użyciem słowa kluczowego jest błędna w wersji 2.x. Nie ma więc prostego sposobu przenoszenia kodu między wersjami. Jednak nie licząc różnic w składni, deklaracja metaklasy w wersjach 2.x i 3.x ma taki sam skutek, czym się za chwilę zajmiemy.

Kierowanie metaklas w wersjach 3.x i 2.x

Po zadeklarowaniu metaklasy w sposób opisany w poprzednim podrozdziale wywołanie tworzące obiekt `class` wykonywane na końcu instrukcji `class` jest modyfikowane, tak by wywołać metaklasę zamiast domyślnej klasy `type`:

```
class = Meta(nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)
```

A ponieważ metaklasa jest klasą podzielną `type`, metoda `__call__` klasy `type` deleguje wywołania tworzące i inicjalizujące nowe obiekty klas do metaklasy, jeśli definiuje ona własne wersje poniższych metod:

```
Meta.__new__(Meta, nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)  
Meta.__init__(klaś, nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)
```

By to zademonstrować, poniżej zaprezentowano ponownie przykład z poprzedniego podrozdziału, rozszerzony o specyfikację metaklasy z wersji 3.x:

```
class Spam(Eggs, metaclass=Meta): # Dziedziczy po Eggs,  
instancja Meta  
  
    data = 1 # Atrybut danych klasy  
  
    def meth(self, arg): # Atrybut metody klasy  
  
        return self.data + arg
```

Na końcu powyższej instrukcji `class` Python wewnętrznie wykonuje poniższy kod w celu utworzenia obiektu `class`. Powtórzę, to wywołanie można wykonać ręcznie, jednak nie jest to konieczne, ponieważ realizuje je mechanizm tworzenia klas w Pythonie.

```
Spam = Meta('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__':  
'__main__'})
```

Jeśli metaklasa definiuje własne wersje metod `__new__` lub `__init__`, zostaną one wywołane w trakcie tego wywołania przez odziedziczoną po `type` metodę `__call__` w celu utworzenia i zainicjalizowania nowej klasy. W efekcie w procesie tworzenia klasy są automatycznie uruchamiane metody metaklasy. W kolejnym podrozdziale pokazane zostanie, jak mogłyby wyglądać kod ostatniej części układanki z metaklasami.



W tym rozdziale stosowana jest składnia z wersji 3.x z użyciem słowa kluczowego, a nie atrybutu jak w wersji 2.x. Ponieważ tworzenie kodu uniwersalnego dla obu wersji nie jest proste, użytkownicy wersji 2.x będą musieli odpowiednio przełożyć kod. W wersji 3.x nie jest rozpoznawana składnia z użyciem atrybutu, a w wersji 2.x z użyciem słowa kluczowego. Z

kolej tworzenie podwójnych listingów nie rozwiązuje problemu kompatybilności kodu (a jedynie wydłuża rozdział!).

Tworzenie metaklas

Dotychczas widzieliśmy, w jaki sposób Python skierowuje tworzenie klasy do metaklasy, o ile taka zostanie podana. W jaki sposób jednak tworzymy metaklasę dostosującą `type` do naszych wymagań?

Okazuje się, że większość zadania jest nam już znana — metaklasy tworzy się za pomocą normalnych instrukcji `class` i semantyki Pythona. Metaklasy są z definicji prostymi klasami pochodnymi od klasy `type`. Jedyna znacząca różnica polega na tym, że Python wywołuje je *automatycznie* na końcu instrukcji `class` i że muszą się one stosować do *interfejsów* oczekiwanych przez klasę nadziedziczoną `type`.

Prosta metaklasa

Chyba najprostsza metaklasa, jaką można utworzyć, to po prostu klasa podzialekna `type` z metodą `__new__` tworzącą obiekt klasy za pomocą wykonania wersji domyślnej w `type`. Taka metoda `__new__` metaklasy wykonywana jest przez metodę `__call__` odziedziczoną po `type`. Zazwyczaj wykonuje pożądane dostosowanie do naszych potrzeb i wywołuje metodę `__new__` klasy nadziedziczonej `type` w celu utworzenia i zwrotienia nowego obiektu klasy:

```
class Meta(type):  
    def __new__(metaklasa, nazwa_klasy, klasy_nadziedziczone, słownik_atrybutów):  
        # Wykonywane przez odziedziczone type.__call__  
        return type.__new__(metaklasa, nazwa_klasy, klasy_nadziedziczone,  
                           słownik_atrybutów)
```

Ta metaklasa tak naprawdę nic nie robi (możemy również dobrze pozwolić domyślnej klasie `type` na utworzenie klasy), jednak demonstruje sposób wejścia metaklasy w miejsce punktu zaczepienia w celu dostosowania klasy. Ponieważ metaklasa wywoływana jest na końcu instrukcji `class`, a metoda `__call__` obiektu `type` odsyła do metod `__new__` oraz `__init__`, kod udostępniony w tych metodach jest w stanie zarządzać wszystkimi klasami utworzonymi z metaklasą.

Oto ponownie nasz przykład w akcji, z dodanymi do metaklasy wywołaniami `print` i plikiem do śledzenia (jak zawsze, również w tym rozdziale w poleceniach użyte są nazwy plików zawierających przykładowe kody):

```
class MetaOne(type):  
    def __new__(meta, classname, supers, classdict):  
        print('W MetaOne.new:', classname, supers, classdict, sep='\n...')  
        return type.__new__(meta, classname, supers, classdict)  
  
class Eggs:  
    pass  
print('tworzenie klasy')
```

```

class Spam(Eggs, metaclass=MetaOne):           # Dziedziczy po Eggs,
    instancji Meta

    data = 1                                     # Atrybut danych klasy

    def meth(self, arg):                         # Atrybut metody klasy
        return self.data + arg

print('tworzenie instancji')
X = Spam()
print('dane:', X.data, X.meth(2))

```

W powyższym kodzie Spam dziedziczy po Eggs i jest instancją MetaOne, natomiast X jest instancją klasy Spam, po której dziedziczy. Po wykonaniu tego kodu w Pythonie 3.x można zauważyc, że metaklasa wywoływana jest na końcu instrukcji class, zanim jeszcze utworzymy instancję — metaklasy służą do przetwarzania klas, natomiast klasy służą do przetwarzania instancji:

```

c:\code> py -3 metaclass1.py
tworzenie klasy
W MetaOne.new:
...<class '__main__.MetaOne'>
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x02A191E0>, '__module__': '__main__'}
tworzenie instancji
dane: 1 3

```

Uwaga do prezentowanych wyników: dla zwięzości opisu adresy zostały przycięte, jak również w wynikach pominięto nieistotne wbudowane nazwy __X__ zawarte w słownikach przestrzeni nazw. Ponadto, jak wspomniano wcześniej, ze względu na inną składnię w wersji 2.x zrezygnowano z dostosowania kodu do tej wersji. Aby uruchomić przykłady w wersji 2.x, należy użyć składni z atrybutem klasy i odpowiednio zmienić instrukcje wyświetlające wyniki. Opisany przykład działa poprawnie w wersji 2.x po wprowadzeniu w pliku *metaclass1-2x.py* poniższych zmian. Należy zwrócić uwagę, że jedna z klas, Eggs lub Spam, musi jawnie dziedziczyć klasę object. W przeciwnym razie pojawi się ostrzeżenie, że klasa w nowym stylu nie może dziedziczyć wyłącznie typowych klas. Aby tego uniknąć, należy w kodzie klienckim korzystającym z metaklasy użyć nazwy object:

```

from __future__ import print_function      # Tylko w wersji 2.x: wiersz
niezbędny do uruchomienia kodu

class Eggs(object):                      # Jeden object opcjonalny

class Spam(Eggs, object):
    __metaclass__ = MetaOne

```

Dostosowywanie tworzenia do własnych potrzeb oraz inicjalizacja

Metaklasy mogą także wejść w miejsce protokołu `__init__` wywołanego przez metodę `__call__` obiektu `type`. Metoda `__new__` tworzy i zwraca obiekt klasy, natomiast `__init__` inicjalizuje już utworzoną, podaną w argumencie klasę. Metaklasy mogą wykorzystać oba punkty zaczepienia do zarządzania klasami w momencie tworzenia:

```
class MetaOne(type):

    def __new__(meta, classname, supers, classdict):
        print('W MetaOne.new: ', classname, supers, classdict, sep='\n... ')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(Class, classname, supers, classdict):
        print('W MetaOne init:', classname, supers, classdict, sep='\n... ')
        print('...obiekt zainicjalizowanej klasy:',
              list(Class.__dict__.keys()))

class Eggs:
    pass

print('tworzenie klasy')

class Spam(Eggs, metaclass=MetaOne):           # Dziedziczy po Eggs,
    instancji Meta

    data = 1                                     # Atrybut danych klasy

    def meth(self, arg):                         # Atrybut metody klasy
        return self.data + arg

print('tworzenie instancji')
X = Spam()
print('dane:', X.data, X.meth(2))
```

W tym przypadku metoda inicjalizacji klasy wykonana zostaje po metodzie tworzenia klasy, a obie wykonywane są na końcu instrukcji `class` przed utworzeniem jakichkolwiek instancji. I odwrotnie: metoda `__init__` klasy `Spam` jest wywoływana w chwili tworzenia jej *instancji*, przy czym nie jest uruchamiana przez metodę `__init__` metaklasy.

```
c:\code> py -3 metaclass2.py
tworzenie klasy
W MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x02967268>, '__module__':
 '__main__'}
W MetaOne init:
...Spam
...(<class '__main__.Eggs'>,)
```

```
...{'data': 1, 'meth': <function Spam.meth at 0x02967268>, '__module__':  
'__main__'}  
...obiekt zainicjalizowanej klasy: ['__qualname__', 'data', '__module__',  
'meth', '__doc__']  
tworzenie instancji  
dane: 1 3
```

Pozostałe sposoby tworzenia metaklas

Choć redefiniowanie metod `__new__` oraz `__init__` klasy nadrzędnej `type` jest najczęstszym sposobem wstawiania logiki do procesu tworzenia obiektu klasy za pomocą metaklasy, możliwe są także inne rozwiązania.

Użycie prostych funkcji fabrycznych

Metaklasy tak naprawdę nie muszą wcale być klasami. Jak już wiemy, instrukcja `class` wykonuje proste wywołanie tworzące klasę na końcu jej przetwarzania. Z tego powodu każdy *obiekt, który można wywoływać*, może być wykorzystany jako metaklasa, pod warunkiem że przyjmuje przekazane argumenty i zwraca obiekt zgodny z zamierzoną klasą. Tak naprawdę prosta funkcja fabryczna obiektu zadziała równie dobrze jak klasa:

```
# Prosta funkcja może także działać jak metaklasa  
  
def MetaFunc(classname, supers, classdict):  
    print('W MetaFunc: ', classname, supers, classdict, sep='\n...')  
    return type(classname, supers, classdict)  
  
class Eggs:  
    pass  
print('tworzenie klasy')  
class Spam(Eggs, metaclass=MetaFunc):          # Wykonanie prostej funkcji  
    na końcu  
        data = 1                                # Funkcja zwraca klasę  
        def meth(self, args):  
            return self.data + arg  
        print('tworzenie instancji')  
X = Spam()  
print('dane:', X.data, X.meth(2))
```

Przy wykonywaniu funkcja wywoływana jest na końcu instrukcji deklarującej klasę i zwraca oczekiwany obiekt nowej klasy. Funkcja ta po prostu przechwytuje wywołanie, które normalnie domyślnie przechwytuje metoda `__call__` obiektu `type`:

```
c:\code> py -3 metaclass3.py  
tworzenie klasy  
W MetaFunc:  
...Spam
```

```

...(<class '__main__.Eggs',>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029471E0>, '__module__':
'__main__'}
tworzenie instancji
dane: 1 3

```

Przeciążenie wywołań tworzących klasę za pomocą zwykłych klas

Ponieważ zwykłe instancje klas mogą po wywołaniu przeciążać operatory, mogą po części pełnić rolę metaklas, podobnie jak opisane wcześniej funkcje. Poniższy kod zwraca podobny wynik jak poprzedni, jednak jest oparty na prostej klasie, która nie pochodzi od klasy `type`, natomiast posiada metodę `__call__` przechwytyującą wywołania metaklasy za pomocą zwykłego przeciążania operatora. Należy zwrócić uwagę, że metody `__new__` i `__init__` muszą mieć różne nazwy. W przeciwnym wypadku będą uruchamiane w chwili *tworzenia* instancji klasy `Meta`, a nie później, gdy klasa ta będzie wywoływana w roli metaklasy.

```

# Zwykła klasa też może pełnić rolę metaklasy
class MetaObj:

    def __call__(self, classname, supers, classdict):
        print('W MetaObj.call: ', classname, supers, classdict, sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class

    def __New__(self, classname, supers, classdict):
        print('W MetaObj.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classdict):
        print('In MetaObj.init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass
print('tworzenie klasy')

class Spam(Eggs, metaclass=MetaObj()):          # MetaObj jest tu instancją
                                                # zwykłej klasy
    data = 1                                     # Operacja wykonywana na końcu
    instrukcji
    def meth(self, arg):
        return self.data + arg
print('tworzenie instancji')
X = Spam()

```

```
print('dane:', X.data, X.meth(2))
```

Po uruchomieniu powyższego kodu wszystkie trzy metody zostaną wykonane w wyniku wywołania metody `__call__` zwykłej instancji, odziedziczonej po zwykłej klasie, bez żadnego uzależnienia od semantyki i mechanizmu wywoływania metod w klasie `type`.

```
c:\code> py -3 metaclass4.py
tworzenie klasy
W MetaObj.call:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029492F0>, '__module__': '__main__'}
W MetaObj.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029492F0>, '__module__': '__main__'}
W MetaObj.init:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x029492F0>, '__module__': '__main__'}
...init class object: ['__module__', '__doc__', 'data', '__qualname__', 'meth']
tworzenie instancji
data: 1 3
```

W rzeczywistości, aby w tym modelu kodowania móc przechwytywać wywołania, można stosować zwykłe dziedziczenie klas. Tutaj klasa nadziedziona pełni w zasadzie taką samą rolę jak klasa `type`, przynajmniej pod względem kierowania metaklas:

```
# Instancje normalnie dziedziczą klasy i ich klasy nadziedzne
class SuperMetaObj:
    def __call__(self, classname, supers, classdict):
        print('In SuperMetaObj.call: ', classname, supers, classdict,
              sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class
class SubMetaObj(SuperMetaObj):
    def __New__(self, classname, supers, classdict):
```

```

        print('In SubMetaObj.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)
    def __Init__(self, Class, classname, supers, classdict):
        print('In SubMetaObj.init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))
    class Spam(Eggs, metaclass=SubMetaObj()): # Wywołanie instancji Sub za pomocą
        Super.__call__
        ...Pozostała część pliku bez zmian...
c:\code> py -3 metaclass4-super.py
tworzenie klasy
W SuperMetaObj.call:
...jak poprzednio...
W SubMetaObj.new:
...jak poprzednio...
W SubMetaObj.init:
...jak poprzednio...
tworzenie instancji
data: 1 3

```

Choć powyższy alternatywny sposób jest poprawny większość metaklas koduje się przez ponowne zdefiniowanie metod `__new__` i `__init__` klasy nadzędnej. W praktyce zapewnia to wystarczającą kontrolę, a często jest to prostsze rozwiązywanie niż inne schematy. Co więcej, metaklasy często mają dostęp do dodatkowych narzędzi (na przykład *metod* klasy, czym zajmiemy się później), dzięki czemu mogą w bardziej bezpośredni sposób wpływać na działanie klasy, niż w przypadku zastosowania innych sposobów.

Jak się przekonamy w dalszej części rozdziału, często prosta metaklasa pełni rolę bardzo podobną do dekoratora klasy, dzięki czemu można ją wykorzystywać do zarządzania zarówno instancjami, jak i klasami. Najpierw jednak zapoznajmy się z przedstawionym w poniższym podrozdziale przykładem wprowadzającym do tematu odwzorowywania nazw metaklas.

Przeciążenie wywołań tworzących klasę za pomocą metaklas

Ponieważ metaklasy biorą udział w normalnych mechanizmach programowania zorientowanego obiektywnego, mogą także przechwytywać *wywołania tworzące klasy* bezpośrednio na końcu instrukcji `class`, redefiniując metodę `__call__` obiektu `type`. Jeżeli klasa ma być ostatecznie utworzona, należy ostrożnie modyfikować metody `__new__` i `__call__`, pamiętając, aby wywoływały swoje odpowiedniki w klasie `type`. Metoda `__call__` musi odwoływać się do klasy `type`, aby uruchomić analogiczne metody w dwóch pozostałych klasach:

```

# Klasa też mogą przechwytywać wywołania klas (ale wbudowane szukają
# metaklas, a nie klas nadzędnych!)
class SuperMeta(type):
    def __call__(meta, classname, supers, classdict):
        print('W SuperMeta.call: ', classname, supers, classdict, sep='\n...')

```

```

        return type.__call__(meta, classname, supers, classdict)
    def __init__(Class, classname, supers, classdict):
        print('W SuperMeta init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))
    print('tworzenie metaklasy')
    class SubMeta(type, metaclass=SuperMeta):
        def __new__(meta, classname, supers, classdict):
            print('W SubMeta.new: ', classname, supers, classdict, sep='\n...')
            return type.__new__(meta, classname, supers, classdict)
        def __init__(Class, classname, supers, classdict):
            print('W SubMeta init:', classname, supers, classdict, sep='\n...')
            print('...obiekt zainicjalizowanej klasy:',
            list(Class.__dict__.keys()))
    class Eggs:
        pass
    print('tworzenie klasy')
    class Spam(Eggs, metaclass=SubMeta): # Wywołanie SubMeta za pomocą
    SuperMeta.__call__
        data = 1
        def meth(self, arg):
            return self.data + arg
    print('tworzenie instancji')
    X = Spam()
    print('dane:', X.data, X.meth(2))

```

Powyższy kod ma swoje osobliwości, które zostaną opisane za chwilę. Po jego uruchomieniu wszystkie zmienione metody klasy Spam są wywoływanie po kolej, jak w poprzednim podrozdziale. Na tym właśnie polega domyślne działanie klasy object. Jednak dodatkowo pojawia się wywołanie podzielnej metaklasy:

```

c:\code> py -3 metaclass5.py
tworzenie klasy
W SuperMeta.call:
...Spam
...(<class '__main__.Eggs'>,
...{'__init__': <function SubMeta.__init__ at 0x028F92F0>, ...}
...init class object: ['__doc__', '__module__', '__new__', '__init__', ...]
W SubMeta.new:

```

```

...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x028F9378>, '__module__': '__main__'}
W SubMeta.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x028F9378>, '__module__': '__main__'}
In SubMeta init:
...Spam
...(<class '__main__.Eggs'>,)
...{'data': 1, 'meth': <function Spam.meth at 0x028F9378>, '__module__': '__main__'}
... obiekt zainicjalizowanej klasy: ['__qualname__', '__module__', '__doc__', 'data', 'meth']
tworzenie instancji
dane: 1 3

```

Powyższy przykład komplikuje fakt, iż przeciążane są metody wywoływane przez *wbudowane* operacje — w tym przypadku automatycznie podczas tworzenia klasy. Metaklasy są wykorzystywane do tworzenia obiektów klas, jednak same nie generują instancji. Z tego powodu w przypadku metaklas reguły wyszukiwania nazw są nieco odmienne od tego, do czego jesteśmy przyzwyczajeni. Metoda `__call__` na przykład wyszukiwana jest w klasie obiektu (`type`). W przypadku metaklas oznacza to metaklasę metaklasy.

Jak się później przekonamy, metaklasy normalnie *dziedziczą* również nazwy z innych metaklas, ale tak jak w przypadku klas dotyczy to tylko *jawnych* pobrań nazw, a nie *niejawnego* wyszukiwania nazw wbudowanych operacji, na przykład wywołań. Ten drugi przypadek dotyczy tylko metaklasy klasy dostępnej w łączu `__class__`, która jest domyślną klasą `type` lub metaklasą. Jest to ten sam problem kierowania wbudowanych operacji, z którym często się w tej książce spotykaliśmy przy okazji zwykłych instancji klas. Metaklasa w klasie `SubMeta` musi ustawać to łącze, choć uruchamia w ten sposób krok konstruowania metaklasy dla samej metaklasy.

Przeanalizujmy początek uzyskanego wyniku. Metoda `__call__` klasy `SuperMeta` *nie* jest uruchamiana podczas tworzenia klasy `SubMeta` (dotyczy to natomiast klasy `type`), ale podczas wywoływania klasy `SubMeta` przy tworzeniu klasy `Spam`. Zwykłe dziedziczenie klasy `SuperMeta` nie wystarcza do przechwytywania wywołań klasy `SubMeta` i z powodów, które poznamy później, nie należy tego robić ze względu na metody przeciążające operatory. Metoda `__call__` klasy `SuperMeta` jest przejmowana przez klasę `Spam`, przez co wywołania tworzące klasę `Spam` nie udają się jeszcze przed utworzeniem instancji. Subtelny problem, jednak problem!

Poniżej znajduje się prostsza ilustracja problemu — zwykła klasa nadrzędna jest pomijana przez *wbudowane* operacje, ale nie przez *jawne* pobrania i wywołania. Te ostatnie opierają się na zwykłym dziedziczeniu nazw atrybutów.

```
class SuperMeta:
```

```

    def __call__(self, classname, supers, classdict):      # Wg nazwy,
niewbudowane

        print('W SuperMeta.call: ', classname, supers, classdict, sep='\n...')

        return type.__call__(meta, classname, supers, classdict)

class SubMeta(SuperMeta):                                # Tworzona
domyślnie przez klasę type

    def __Init__(Class, classname, supers, classdict):    # Nadpisuje metodę
__init__ klasy type

        print('W SubMeta init:', classname)

print(SubMeta.__class__)

print([n.__name__ for n in SubMeta.__mro__])

print()

print(SubMeta.__call__)                                # Brak deskryptora
danych, jeżeli znalezione po nazwie

print()

SubMeta.__call__(SubMeta, 'xxx', (), {})               # Jawne wywołanie
działa: dziedziczenie klasy

print()

SubMeta('yyy', (), {})                               # Jednak niejawne
wywołanie wbudowanej operacji nie działa

c:\code> py -3 metaclass5b.py

<class 'type'>

['SubMeta', 'SuperMeta', 'type', 'object']

<function SuperMeta.__call__ at 0x029B9158>

W SuperMeta.call: xxx

W SubMeta init: xxx

W SubMeta init: yyy

```

Oczywiście ten specyficzny przykład jest przypadkiem specjalnym: przechwytywanie wbudowanych operacji w metaklasie stosuje się rzadziej niż metodę `__call__`. Pokazuje jednak zasadniczą asymetrię i pozorną niespójność: *zwykłe dziedziczenie atrybutu nie jest w pełni wykorzystywane do kierowania wbudowanymi operacjami*, zarówno w instancjach, jak i w klasach.

Aby w pełni zrozumieć niuanse tego przykładu, musimy się dokładniej dowiedzieć, jaką rolę odgrywają metaklasy w procesie odwzorowywania nazw w Pythonie.

Instancje a dziedziczenie

Ponieważ metaklasy określane są w sposób podobny do klas nadrzędnych dziedziczenia, na pierwszy rzut oka mogą być nieco mylące. Kilka ważnych uwag powinno pomóc podsumować i

wyjaśnić ten model:

- **Metaklasy dziedziczą po klasie type.** Choć pełnią specjalną rolę, metaklasy tworzone są za pomocą instrukcji `class` i zachowują się zgodnie ze zwykłym modelem programowania zorientowanego obiektowo w Pythonie. Przykładowo jako klasy podrzędne `type` mogą redefiniować metody obiektu `type`, w miarę potrzeby nadpisując je i dostosowując do własnych wymagań. Metaklasy zazwyczaj redefiniują metody `__new__` i `__init__` klasy `type` w celu dostosowania tworzenia klasy oraz inicjalizacji do własnych potrzeb, jednak mogą także redefiniować metodę `__call__`, jeśli chcą bezpośrednio przechwytywać wywołanie tworzące klasę na jej końcu (pomimo złożoności opisanych w poprzedniej części rozdziału). Choć jest to rzadziej spotykane, mogą nawet być prostymi funkcjami zwracającymi dowolne obiekty, a nie tylko klasami podrzędnymi `type`.
- **Deklaracje metaklas dziedziczone są przez klasy podrzędne.** Deklaracja `metaclass=M` w klasie definiowanej przez użytkownika `dziedziczona` jest także przez jej klasy podrzędne, przez co metaklasa zostanie wykonana w trakcie tworzenia każdej klasy dziedziczącej tę specyfikację w łańcuchu klas nadrzędnych.
- **Atrybuty metaklas nie są dziedziczone przez instancje klas.** Deklaracje metaklas określają relację `instancji`, która nie jest tym samym, co dziedziczenie. Ponieważ klasy są instancjami metaklas, działanie zdefiniowane w metaklasie ma zastosowanie do klasy, jednak nie do późniejszych instancji klasy. Instancje pozyskują zachowanie ze swoich klas i klas nadrzędnych, jednak nie z metaklas. Wyszukiwanie atrybutów instancji przeszukuje jedynie słowniki `__dict__` instancji i wszystkich jej klas. Metaklasa *nie* jest zawarta w wyszukiwaniu dziedziczenia.
- **Atrybuty metaklasy są przejmowane przez klasy.** Klasy przejmują metody swoich metaklas na mocy relacji instancyjnej. Jest to źródło działania klas przetwarzających same klasy. Klasy przejmują atrybuty metaklas za pośrednictwem łącza `__class__` w taki sam sposób, jak zwykłe instancje przejmują nazwy od swoich klas. Jednak wcześniej podejmowana jest próba dziedziczenia poprzez przeszukiwanie słownika `__dict__`. Jeżeli ta sama nazwa jest dostępna dla klasy *zarówno* w metaklasie, jak i klasie nadrzędnej, wtedy wybierana jest druga opcja. Jednak atrybut `__class__` nie jest wykorzystywany przez własne instancje. Atrybuty metaklasy są udostępniane klasom instancji, ale nie instancjom.

Być może kod będzie bardziej zrozumiałym niż opis. By zilustrować wszystkie powyższe punkty, rozważmy poniższy przykład:

```
# Plik metainstance.py

class MetaOne(type):

    def __new__(meta, classname, supers, classdict): # Redefiniuje metodę
        klasy type
            print('W MetaOne.new:', classname)
            return type.__new__(meta, classname, supers, classdict)

    def toast(self):
        print('tost')

class Super(metaclass=MetaOne):      # Metaklasy dziedziczone także przez klasy
    podrzędne

        def spam(self):          # MetaOne wykonana 2 razy dla 2 klas
            print('mielonka')

class C(Super):                    # Klasa nadrzędna: dziedziczenie a
    instancja
```

```
def eggs(self):                      # Klasa dziedziczą po klasach nadrzędnych
    print('jajka')                  # Ale nie po metaklasach
```

Po wykonaniu powyższego kodu metaklasa obsługuje tworzenie *obu* klas klienta, a *instancje* dziedziczą atrybuty klas, jednak atrybutów metaklasy już *nie*:

```
>>> from metainstance import *      # Wykonanie instrukcji class: metaklasa
jest wykonywana dwukrotnie

W MetaOne.new: Super
W MetaOne.new: Sub

>>> X = Sub()                      # Zwykła instancja klasy zdefiniowanej
przez użytkownika

>>> X.eggs()                       # Metoda odziedziczona po Sub
'jajka'

>>> X.spam()                        # Metoda odziedziczona po Super
'mielonka'

>>> X.toast()                       # Metoda nieodziedziczona po metaklasie
AttributeError: 'Sub' object has no attribute 'toast'
```

Natomiast klasy dziedziczą nazwy po klasach nadrzędnych, jak również przejmują je od swojej metaklasy (która w tym przykładzie *również* pochodzi od klasy nadrzędnej):

```
>>> Sub.eggs(X)                   # Własna metoda
'jajka'

>>> Sub.spam(X)                   # Metoda odziedziczona po Super
'mielonka'

>>> Sub.toast()                   # Metoda przejęta od metaklasy
'tost'

>>> Sub.toast(X)                  # To nie jest zwykła metoda
TypeError: toast() takes 1 positional argument but 2 were given
```

Należy zwrócić uwagę, że ostatnie wywołanie w powyższym przykładzie zakończyło się niepowodzeniem, ponieważ nazwa została odwzorowana na metodę metaklasy, a nie zwykłej klasy. W rzeczywistości zarówno obiekt, z którego pobierana jest nazwa, jak i jego źródło mają tu krytyczne znaczenie. Metody przejęte od metaklas są powiązane z przedmiotową *klasą*, natomiast metody ze zwykłych klas *nie są powiązane*, jeżeli są pobierane poprzez klasę, za to są *powiązane* podczas pobierania ich poprzez instancję:

```
>>> Sub.toast
<bound method MetaOne.toast of <class 'metainstance.Sub'>>
>>> Sub.spam
<function Super.spam at 0x0298A2F0>
>>> X.spam
<bound method Sub.spam of <metainstance.Sub object at 0x02987438>>
```

Dwie ostatnie reguły przeanalizowaliśmy wcześniej w rozdziale 31. poświęconym powiązonym metodom. Pierwsza reguła jest nowa, ale przypomina metody klasy. Aby zrozumieć, dlaczego działa to w taki, a nie inny sposób, musimy dokładniej poznać związki z instancjami metaklas.

Metaklasa a klasa nadrzędna

Aby rzecz jeszcze bardziej uprościć, sprawdźmy, co się stanie w następującym przypadku: klasa B, będąca *instancją* metaklasy A, przejmuje jej atrybut, który nie jest dostępny dla instancji pochodnych od klasy B. Przejmowanie nazw od instancji metaklasy *różni się* od zwykłego dziedziczenia klas instancji:

```
>>> class A(type): attr = 1
>>> class B(metaclass=A): pass          # Klasa B jest instancją metaklasy i
                                         przejmuje jej atrybut
>>> I = B()                           # Instancja I pochodzi od klasy, a nie
                                         metaklasy!
>>> B.attr
1
>>> I.attr
AttributeError: 'B' object has no attribute 'attr'
>>> 'attr' in B.__dict__, 'attr' in A.__dict__
(False, True)
```

Natomiast jeżeli A przemieni się z metaklasy w klasę nadrzędną, wtedy nazwy *oddziedziczone* z klasy A będą dostępne w tworzonych później instancjach klasy B i będą wyszukiwane w słowniku przestrzeni nazw w klasach w drzewie. Oznacza to, że obiekty będą wyszukiwane w słowniku `__dict__` w zadanej kolejności odwzorowywania metod (ang. *method resolution order* – MRO) tak samo jak w funkcji `mapattrs` w przykładzie opisany w rozdziale 32.:

```
>>> class A: attr = 1
>>> class B(A): pass                  # Instancja I dziedziczy klasę
>>> I = B()
>>> B.attr
1
>>> I.attr
1
>>> 'attr' in B.__dict__, 'attr' in A.__dict__
(False, True)
```

Z tego powodu metaklasy, chcąc zmienić działanie tworzonych później instancji, często wykonują swoją robotę, manipulując słownikiem przestrzeni nazw nowej klasy. Instancje wtedy „widzą” nazwy zdefiniowane w klasie, ale nie w metaklasie. Zobaczmy jednak, co się stanie, jeżeli ta sama nazwa będzie dostępna w *obu* źródłach atrybutów. Wtedy nazwa zostanie *oddziedziczona*, a nie przejęta:

```
>>> class M(type): attr = 1
```

```

>>> class A: attr = 2
>>> class B(A, metaclass=M): pass      # Klasa nadzędna ma większy priorytet
niż metaklasa
>>> I = B()
>>> B.attr, I.attr
(2, 2)
>>> 'attr' in B.__dict__, 'attr' in A.__dict__, 'attr' in M.__dict__
(False, True, True)

```

Ta zasada obowiązuje niezależnie od względnej wysokości hierarchii dziedziczenia i źródeł instancji. Najpierw jest sprawdzany słownik `__dict__` każdej klasy zgodnie z MRO (dziedziczenie), a dopiero potem następuje przejmowanie metaklas (*instancje*):

```

>>> class M(type): attr = 1
>>> class A: attr = 2
>>> class B(A): pass
>>> class C(B, metaclass=M): pass      # Klasa nadzędna znajduje się dwa
poziomy wyżej nad metaklasą, ale i tak wygrywa
>>> I = C()
>>> I.attr, C.attr
(2, 2)
>>> [x.__name__ for x in C.__mro__]    # Patrz szczegółowy opis MRO w
rozdziale 32.
['C', 'B', 'A', 'object']

```

W rzeczywistości klasy przejmują atrybuty metaklasy za pośrednictwem łącza `__class__` w taki sam sposób, w jaki instancje dziedziczą adresy po klasach za pomocą własnych łączy `__class__`. Ma to sens, zważywszy, że klasy są również instancjami metaklas. Zasadnicza różnica polega na tym, że dziedziczenie instancji nie odbywa się za pomocą łącza `__class__`, tylko ograniczony jest jego zasięg do słownika `__dict__` każdej klasy w drzewie zgodnie z MRO. Wykorzystywany jest przy tym tylko atrybut `__bases__` każdej klasy i łącze `__class__` instancji:

```

>>> I.__class__                  # Wykorzystywane podczas dziedziczenia:
instancja klasy
<class '__main__.C'>
>>> C.__bases__                 # Wykorzystywane podczas dziedziczenia: klasa
nadzędna
(<class '__main__.B'>,)
>>> C.__class__                  # Wykorzystywane podczas przejmowania instancji:
metaklasa
<class '__main__.M'>
>>> C.__class__.attr            # Inny sposób uzyskania atrybutów metaklasy

```

Po przeanalizowaniu powyższego kodu prawdopodobnie widoczna będzie wyraźna symetria, która jest tematem następnego podrozdziału.

Dziedziczenie: pełna historia

Jak się okazuje, dziedziczenie instancji odbywa się w podobny sposób, niezależnie od tego, czy „instancja” jest tworzona na podstawie zwykłej klasy, czy klasa jest tworzona na podstawie metaklasy klasy pochodnej od `type`. Jest to jedna, prosta zasada wyszukiwania atrybutów, zgodna z szeroko pojętą hierarchią dziedziczenia metaklas. W poniższym kodzie ilustrującym podstawy tego koncepcjonalnego połączenia instancja dziedziczy nazwy po wszystkich klasach, klasa dziedziczy zarówno po klasach, jak i metaklasach, a metaklasy dziedziczą po wyższych metaklasach (metaklasach nadrzędnych):

```
>>> class M1(type): attr1 = 1          # Drzewo dziedziczenia  
metaklas  
  
>>> class M2(M1): attr2 = 2          # Uzyskanie __bases__,  
__class__, __mro__  
  
>>> class C1: attr3 = 3              # Drzewo dziedziczenia klas  
  
>>> class C2(C1,metaclass=M2): attr4 = 4    # Uzyskanie __bases__,  
__class__, __mro__  
  
>>> I = C2()                      # Instancja I uzyskuje  
__class__, ale nic poza tym  
  
>>> I.attr3, I.attr4            # Instancja dziedziczy po  
nadklasach w drzewie  
  
(3, 4)  
  
>>> C2.attr1, C2.attr2, C2.attr3, C2.attr4      # Klasa uzyskuje nazwy z obu  
drzew!  
  
(1, 2, 3, 4)  
  
>>> M2.attr1, M2.attr2            # Metaklasy również  
dziedziczą nazwy!  
  
(1, 2)
```

Obie ścieżki dziedziczenia — po klasach i metaklasach — wykorzystują te same łączą, jednak nie rekurencyjnie. Instancje nie dziedziczą nazw po swoich metaklasach, ale mogą ich jawnie żądać:

```
>>> I.__class__                  # Łączą wykorzystane przez  
instancję bez __bases__  
  
<class '__main__.C2'>  
  
>>> C2.__bases__                # Łączą wykorzystane przez klasę po  
__bases__  
  
<class '__main__.M2'>  
  
>>> M2.__bases__  
  
(<class '__main__.M1'>,)
```

```

>>> I.__class__.attr1          # Skierowanie dziedziczenia do
drzewa metaklas

1

>>> I.attr1                  # Łącze __class__ nie jest
wykorzystywane normalnie

AttributeError: 'C2' object has no attribute 'attr1'

>>> M2.__class__              # Oba drzewa mają MRO i łącza
instancji

<class 'type'>

>>> [x.__name__ for x in C2.__mro__]      # Drzewo __bases__ z I.__class__
['C2', 'C1', 'object']

>>> [x.__name__ for x in M2.__mro__]      # Drzewo __bases__ z C2.__class__
['M2', 'M1', 'type', 'object']

```

Jeżeli metaklasy są ważne lub stosowany jest kod, który ich wymaga, warto dokładnie przeanalizować powyższe przykłady. W dziedziczeniu najpierw wykorzystywany jest atrybut `__bases__`, a po nim jeden atrybut `__class__`. Zwykłe instancje nie mają atrybutu `__bases__`, natomiast klasy (zarówno zwykłe, jak i metaklasy) mają oba. W rzeczywistości dokładne poznanie powyższych przykładów jest niezbędne do zrozumienia algorytmu odwzorowywania nazw, o którym mowa w następnym podrozdziale.

Algorytm dziedziczenia w Pythonie: wersja prosta

Teraz gdy znany jest proces przejmowania nazw z metaklas, można sformułować reguły dziedziczenia, które ten proces rozszerza. Dziedziczenie opiera się na algorytmie MRO i obejmuje dwie osobne, ale podobnie wyglądające procedury przeszukujące. Ponieważ atrybut `__bases__` jest wykorzystywany w trakcie wykonywania kodu do konstruowania atrybutu `__mro__` w chwili tworzenia klasy, a atrybut ten zawiera samego siebie, opisane w poprzednim podrozdziale uogólnienie jest takie samo jak przedstawiona niżej pierwsza definicja algorytmu dziedziczenia w nowym stylu w Pythonie.

W celu znalezienia jawnnej nazwy atrybutu wykonaj następujące operacje:

1. Przeszukaj *instancję* I, następnie jej klasę i wszystkie klasy nadzędne, wykorzystując:
 - a. słownik `__dict__` instancji I,
 - b. słownik `__dict__` wszystkich klas w atrybucie `__mro__` znalezionym w atrybucie `__class__` instancji I, w kolejności od lewej do prawej.
2. Przeszukaj *klasę* C, następnie wszystkie jej klasy nadzędne i drzewo metaklas, wykorzystując:
 - a. słownik `__dict__` wszystkich klas w atrybucie `__mro__` znalezionym w klasie C, w kolejności od lewej do prawej,
 - b. słownik `__dict__` wszystkich metaklas w atrybucie `__mro__` znalezionym w klasie C, w kolejności od lewej do prawej.
3. W regułach 1 i 2 nadaj pierwszeństwo *deskryptoram danych* znalezionym w źródłach w kroku b (patrz dalej).

4. W regułach 1 i 2 pomiń krok a i zacznij wyszukiwanie *wbudowanych* operacji od kroku b (patrz dalej).

Pierwsze dwa kroki wykonywane są tylko w przypadku jawnego pobierania zwykłych atrybutów. Wyjątek stanowią operacje wbudowane i deskryptory, które będą opisane za chwilę. Dodatkowo w przypadku brakujących i wszystkich nazw można wykorzystać odpowiednio metody `__getattr__` i `__getattribute__` w sposób opisany w rozdziale 38.

Dla większości programistów ważna jest znajomość pierwszej reguły i pierwszego kroku drugiej, które razem odpowiadają *zwykłemu dziedziczeniu klas* w wersji 2.x. Dodatkowy krok przejmowania nazw (2b) został wprowadzony dla metaklas, ale jest on w zasadzie taki sam jak inne kroki. Jest to, trzeba przyznać, dość subtelna równoważność, lecz przejmowanie metaklas nie jest takie nowatorskie, jak się wydaje. W rzeczywistości jest to jeden z komponentów większego modelu.

Specjalny przypadek deskryptorów

Przypadek deskryptorów jest normalny i *uproszczony*. W poprzednim podrozdziale specjalnie został dodany krok 3, choć nie dotyczy on większości kodów i istotnie komplikuje algorytm. Okazuje się jednak, że jest specjalny przypadek interakcji dziedziczenia z deskryptorami atrybutów opisanymi w rozdziale 38. W skrócie: tzw. *deskryptory danych* definiujące metody `__set__` przechwytyjące przypisywanie mają pierwszeństwo, przez co ich nazwy zastępują inne źródła dziedziczenia.

Ten wyjątek odgrywa kilka praktycznych ролей. Na przykład jest wykorzystywany do sprawdzania, czy atrybuty `__class__` i `__dict__` nie mogą być ponownie zdefiniowane za pomocą tej samej nazwy znajdującej się we własnym atrybutie `__dict__` instancji:

```
>>> class C: pass                                # Specjalny przypadek dziedziczenia
nr 1...
>>> I = C()                                         # Deskryptory danych mają
pierwszeństwo
>>> I.__class__, I.__dict__
(<class '__main__.C', {})
>>> I.__dict__['name'] = 'robert'                  # Dynamiczne dane w instancji
>>> I.__dict__['__class__'] = 'spam'               # Przypisanie kluczy, a nie atrybutów
>>> I.__dict__['__dict__'] = {}
>>> I.name                                         # I.name pochodzi z I.__dict__ jak
zwykle,
'robert'                                           # ale nie I.__class__ ani
I.__dict__!
>>> I.__class__, I.__dict__
(<class '__main__.C', {'__class__': 'mielonka', '__dict__': {}, 'name':
'robert'})
```

Tej wyjątek dotyczący deskryptora danych jest sprawdzany w ramach wstępnego kroku wykonywanego przed dwiema poprzednimi regułami i może być ważniejszy dla implementatorów Pythona niż dla programistów. W większości aplikacji można go bez obaw pominąć, chyba że kodowane są własne deskryptory danych, w przypadku których obowiązuje ta sama zasada specjalnego przypadku pierwszeństwa dziedziczenia:

```
>>> class D:
```

```

        def __get__(self, instance, owner): print('__get__')
        def __set__(self, instance, value): print('__set__')

>>> class C: d = D()                      # Atrybut deskryptora danych
>>> I = C()
>>> I.d                                    # Odziedziczony dostęp do atrybutu
danych

__get__
>>> I.d = 1

__set__
>>> I.__dict__['d'] = 'mielonka'          # Definicja tej samej nazwy w słowniku
przestrzeni nazw instancji
>>> I.d                                    # Jednak nie ukrywa to deskryptora
danych w klasie!

__get__

```

I odwrotnie, jeżeli deskryptor nie zdefiniuje atrybutu `__set__`, wtedy nazwa w słowniku instancji przeszła nazwę w klasie w toku zwykłego dziedziczenia:

```

>>> class D:
        def __get__(self, instance, owner): print('__get__')

>>> class C: d = D()
>>> I = C()
>>> I.d                                    # Dostęp do odziedziczonego deskryptora
niedanych

__get__
>>> I.__dict__['d'] = 'mielonka'          # Przesłonienie nazw klas zgodnie z
regułami zwykłego dziedziczenia
>>> I.d
'mielonka'

```

W obu przypadkach zamiast zwracać sam obiekt deskryptora, Python automatycznie wywołuje metodę `__get__` deskryptora, gdy zostanie znaleziona w procesie dziedziczenia. Jest to część magii atrybutów, o której była mowa wcześniej. Jednak specjalny status przyznany deskryptorom zmienia znaczenie *dziedziczenia* atrybutów, a przez to znaczenie nazw w kodzie.

Algorytm dziedziczenia w Pythonie: wersja nieco pełniejsza

Znając specjalny przypadek deskryptora danych i ogólne wywołanie deskryptora uwzględnione w drzewach klas i metaklas, pełny algorytm dziedziczenia w nowym stylu w Pythonie można sformułować w opisany niżej sposób. Jest to wprawdzie skomplikowana procedura, zakładająca znajomość deskryptorów, metaklas i MRO, ale stanowi ostateczną wykładnię odwzorowywania nazw (w poniższym opisie operacje są wykonywane w kolejności zgodnej z numeracją lub od lewej do prawej w warunkach „lub”).

W celu znalezienia jawniej nazwy atrybutu wykonaj następujące operacje:

1. Przeszukaj *instancję* I, a potem jej klasę i wszystkie klasy nadrzędne w następujący sposób:
 - a. Przeszukaj słownik `__dict__` wszystkich klas w atrybucie `__mro__` znalezionym w atrybucie `__class__` instancji I.
 - b. Jeżeli deskryptor danych został znaleziony w kroku a, wywołaj jego metodę `__get__` i wyjdź.
 - c. W przeciwnym wypadku zwróć wartość znalezioną w słowniku `__dict__` instancji I.
 - d. W przeciwnym wypadku wywołaj deskryptor niedanych lub zwróć wartość znalezioną w kroku a.
2. Przeszukaj *klasę* C, a potem wszystkie jej klasy nadrzędne i drzewo metaklas w następujący sposób:
 - a. Przeszukaj słownik `__dict__` wszystkich metaklas w atrybucie `__mro__` znalezionym w atrybucie `__class__` klasy C.
 - b. Jeżeli deskryptor danych został znaleziony w kroku a, wywołaj jego metodę `__get__` i wyjdź.
 - c. W przeciwnym wypadku wywołaj deskryptor lub zwróć wartość znalezioną w słowniku `__dict__` z własnego atrybutu `__mro__` klasy C.
 - d. W przeciwnym wypadku wywołaj deskryptor niedanych lub zwróć wartość znalezioną w kroku a.
3. W regułach 1 i 2 w przypadku *wbudowanych* operacji wykorzystaj jedynie źródła wymienione w kroku a (patrz dalej).

Należy ponownie zwrócić uwagę, że procedura ta dotyczy zwykłego, *jawnego* pobierania atrybutów. Powyższe reguły nie dotyczą niejawnego wyszukiwania nazw metod wykorzystywanych przez *wbudowane* operacje. W takiej sytuacji w obu przypadkach wykorzystywane są tylko źródła wymienione w kroku a, co demonstruje następny podrozdział.

Jak zawsze, niejawną klasą nadrzędną obiektu oferuje domyślne dane znajdujące się na szczytce każdego drzewa klas i metaklas (tj. na końcu każdego MRO). Ponadto metoda `__getattr__`, jeżeli jest zdefiniowana, może być wywoływana, gdy atrybut nie zostanie znaleziony, a metoda `__getattribute__` może być wywoływana przy każdym pobraniu atrybutu, choć są to specjalne rozszerzenia modelu wyszukiwania nazw. Więcej informacji o tych narzędziach i deskryptorach jest zawartych w rozdziale 38., natomiast w rozdziale 32. opisane jest specjalne skanowanie klas nadrzędnych w MRO.

Dziedziczenie przypisań atrybutów

Należy zwrócić uwagę, że w poprzednim podrozdziale dziedziczenie zostało zdefiniowane w odniesieniu do *referencji* (wyszukiwania) atrybutów, natomiast po części obejmuje ono również *przypisanie* atrybutów. Jak wiemy, przypisania zmieniają atrybuty przedmiotowego obiektu, jednak dziedziczenie jest wywoływanie dla atrybutów, aby najpierw sprawdzić niektóre opisane w rozdziale 38. narzędzia do zarządzania atrybutami, w tym deskryptory i właściwości. Narzędzia te, jeżeli istnieją, przechwytyują przypisania atrybutów i mogą nimi dowolnie kierować.

Na przykład, jeżeli przypisanie atrybutu jest wykonywane w klasie w nowym stylu, deskryptor danych w nowym stylu z metodą `__set__` jest przejmowany od klasy w wyniku dziedziczenia

zgodnie z MRO i ma pierwszeństwo przed normalnym modelem składowania. Zgodnie z zasadami z poprzedniego podrozdziału:

- W takim przypisaniu dotyczącym *instancji* wykonywane są kroki reguły 1 od a do c przeszukujące drzewo klas instancji. Jednak w kroku b wywoływana jest metoda `_set__`, a nie `_get__`, natomiast w kroku c procedura jest zatrzymywana i atrybut nie jest pobierany, tylko zapisywany w instancji.
- W takim przypisaniu dotyczącym *klasy* wykonywana jest ta sama procedura w drzewie metaklas, podobna do reguły 2, z tą różnicą, że w kroku c procedura jest zatrzymywana, a atrybut zapisywany w klasie.

Ponieważ deskryptory stanowią również podstawę dla innych zaawansowanych narzędzi dotyczących atrybutów, takich jak właściwości i sloty, to wstępne sprawdzenie dziedziczenia przypisania jest wykorzystywane w wielu kontekstach. W efekcie deskryptory są w klasach w nowym stylu traktowane jako specjalne przypadki *zarówno* odwołania, jak i przypisania.

Specjalny przypadek wbudowanych operacji

Jak widzieliśmy, opisane reguły nie dotyczą wbudowanych operacji. Instancje i klasy mogą być pominięte tylko w przypadku wbudowanych operacji jako specjalny przypadek różniący się od zwykłego i jawnego dziedziczenia nazw. Ponieważ jest to rozbieżność specyficzna dla *kontekstu*, łatwiej będzie zademonstrować ją na przykładzie kodu niż w postaci jednego algorytmu. W poniższym przykładzie `str` jest wbudowaną funkcją, `__str__` jest jej jawnym odpowiednikiem, a instancja jest pomijana tylko w przypadku wbudowanej operacji:

```
>>> class C:                                     # Specjalny przypadek
    dziedziczenia nr 2...
        attr = 1                                    # Wbudowana operacja pomija
    krok a
        def __str__(self): return('class')
>>> I = C()
>>> I.__str__(), str(I)                         # Obie nazwy z klasy, jeżeli
    nie ma ich w instancji
('class', 'class')
>>> I.__str__ = lambda: 'instance'
>>> I.__str__(), str(I)                         # Jawnia => instancja,
    wbudowana => klasa!
('instance', 'class')
>>> I.attr                                     # Asymetria ze zwykłymi lub
    jawnymi nazwami
1
>>> I.attr = 2; I.attr
2
```

Jak widzieliśmy wcześniej w przykładzie *metaclass5.py*, tak samo jest w przypadku *klas*: jawnie nazwy zaczynają się w klasie, a wbudowane w klasie klasy, czyli jej metaklasie, domyślnie `type`:

```
>>> class D(type):
    def __str__(self): return('D class')
```

```

>>> class C(D):
        pass

>>> C.__str__(C), str(C)                                # Jawną => klasa nadziedziczeniowa,
wbudowana => metaklasa!

('D class', '<class \'__main__.C\'>')

>>> class C(D):
        def __str__(self): return('C class')

>>> C.__str__(C), str(C)                                # Jawną => klasa nadziedziczeniowa,
wbudowana => metaklasa!

('C class', '<class \'__main__.C\'>')

>>> class C(metaclass=D):
        def __str__(self): return('C class')

>>> C.__str__(C), str(C)                                # Wbudowana => metaklasa
zdefiniowana przez użytkownika

('C class', 'D class')

```

W rzeczywistości czasami uzyskanie informacji, skąd pochodzi nazwa, nie jest w tym modelu prostym zadaniem, ponieważ wszystkie klasy pochodzą od klasy `object`, w tym również domyślna metaklasa `type`. W poniższym jawnym wywołaniu wydaje się, że klasa `C` pobiera domyślną metodę `__str__` z klasy `object`, a nie z metaklasy, zgodnie z pierwszym źródłem dziedziczenia klasy (własnego MRO klasy). Natomiast wbudowane operacje przechodzą od razu do metaklasy, jak poprzednio:

```

>>> class C(metaclass=D):
        pass

>>> C.__str__(C), str(C)                                # Jawną => object, wbudowana =>
metaklasa

('<class \'__main__.C\'>', 'D class')

>>> C.__str__
<slot wrapper '__str__' of 'object' objects>
>>> for k in (C, C.__class__, type): print([x.__name__ for x in k.__mro__])
['C', 'object']
['D', 'type', 'object']
['type', 'object']

```

Wszystko to prowadzi nas do finałowego cytatu tej książki — zasady, która wydaje się kolidować ze statusem nadanym deskryptorom i wbudowanym operacjom w mechanizmie dziedziczenia atrybutów klas w nowym stylu:

Przypadki specjalne nie są wystarczająco specjalne, aby łamały zasady.

Oczywiście, z pewnych praktycznych względów trzeba wprowadzać wyjątki. Zrezygnujemy tutaj z uzasadnień, jednak należy dokładnie rozważyć konsekwencje używania języka obiektowego, w którym *podstawowa operacja*, jaką jest dziedziczenie, jest stosowana w tak nierówny i niespójny sposób. Powinno to przynajmniej podkreślić znaczenie zachowania prostoty kodu, aby

nie uzależniać go od tak zawiłych zasad. Wtedy użytkownicy i administratorzy Twojego kodu będą zadowoleni.

Aby dokładniej poznać całą historię, warto zapoznać się z wewnętrzna implementacją dziedziczenia — aktualna pełna „saga” jest zawarta w plikach *object.c* i *typeobject.c*. Pierwszy z nich obejmuje zwykłe instancje, a drugi klasy. Aby używać Pythona, nie trzeba wprawdzie zagłębiać się w jego wewnętrzne tajniki, ale jest to ostateczne i najlepsze, jakie można znaleźć, źródło prawdy o tym skomplikowanym i rozwijającym się systemie. Dotyczy to szczególnie przypadków specjalnych, które wykształciły się z wyjątków. My natomiast zajmijmy się ostatnią częścią magii metaklas.

Metody metaklas

Równie ważne jak dziedziczenie nazw jest przetwarzanie przez metaklasy *klas* instancji, przy czym nie dotyczy to zwykłych obiektów instancji, ale samych klas. Pod tym względem metaklasy są podobne do *metod klas* opisanych w rozdziale 32., jednak są one dostępne tylko w dziedzinie instancji metaklas, a nie normalnego dziedziczenia instancji. Na przykład błąd na końcu poniższego przykładu wynika z opisanych w poprzednim podrozdziale reguł dziedziczenia jawnego nazw:

```
>>> class A(type):
        def x(cls): print('ax', cls)                      # Metaklasa A (instancje =
        klasy)
        def y(cls): print('ay', cls)                      # Metoda y jest
        nadpisywana przez instancję B
>>> class B(metaclass=A):
        def y(self): print('by', self)                  # Zwykła klasa A (zwykłe
        instancje)
        def z(self): print('bz', self)                  # Słownik przestrzeni nazw
        zawiera metody y i z
>>> B.x                                         # Metoda x przejęta od
metaklasy
<bound method A.x of <class '__main__.B'>>
>>> B.y                                         # Metody y i z
zdefiniowane w samej klasie
<function B.y at 0x0295F1E0>
>>> B.z
<function B.z at 0x0295F378>
>>> B.x()                                       # Wywołanie metody
metaklasy: pobranie cls
ax <class '__main__.B'>
>>> I = B()                                     # Wywołanie klasy
instancji: pobranie inst
>>> I.y()
```

```

by <__main__.B object at 0x02963BE0>
>>> I.z()
bz <__main__.B object at 0x02963BE0>
>>> I.x()                                     # Instancja nie 'widzi'
nazw metaklasy
AttributeError: 'B' object has no attribute 'x'

```

Metody metaklasy a metody klasy

Metody metaklasy, które podobnie jak metody klasy różnią się widocznością w dziedziczeniu, są przeznaczone do zarządzania *danymi na poziomie klasy*. W rzeczywistości ich role zachodzą na siebie, podobnie jak w przypadku metaklas i dekoratorów klas. Jednak metody metaklasy są dostępne jedynie poprzez klasę i, aby zostały powiązane z klasą, nie wymagają użycia jawniej deklaracji `classmethod` na poziomie danych. Innymi słowy, metody metaklasy można traktować jak niejawne metody klasy o ograniczonej widoczności:

```

>>> class A(type):
        def a(cls):                      # Metoda metaklasy: uzyskuje klasę
            cls.x = cls.y + cls.z

>>> class B(metaclass=A):
        y, z = 11, 22
        @classmethod                      # Metoda klasy: uzyskuje klasę
        def b(cls):
            return cls.x

>>> B.a()                           # Wywołanie metody metaklasy: widoczne
tylko dla klasy

>>> B.x                            # Utworzenie danych klasy B, dostępnych
dla zwykłych instancji
33

>>> I = B()
>>> I.x, I.y, I.z
(33, 11, 22)

>>> I.b()                           # Metoda klasy: wysyła klasę, a nie
instancję; widoczna dla instancji
33

>>> I.a()                           # Metody metaklasy: dostępne tylko
poprzez klasę
AttributeError: 'B' object has no attribute 'a'

```

Przeciążanie operatorów w metodach metaklasy

Metaklasy, podobnie jak zwykłe klasy, mogą przeciągać operatory, aby na instancjach klas można było wykonywać wbudowane operacje. Na przykład metoda indeksująca `__getitem__` w poniższej metaklasie służy do przetwarzania samych *klas*, tj. instancji tej metaklasy, a nie tworzonych później instancji jej klas. W rzeczywistości, zgodnie z opisany wcześniejszym algorytmem, instancje zwykłych klas w ogóle nie dziedziczą nazw przejętych za pośrednictwem relacji między instancjami metaklasy, choć mają dostęp do nazw znajdujących się w ich własnych klasach:

```
>>> class A(type):
    def __getitem__(cls, i):      # Metoda metaklasy do przetwarzania
        klas
        return cls.data[i]         # Wbudowana operacja pomija klasę,
        należy użyć metaklasy
        # Wyszukiwanie jawnych nazw w klasie i
        metaklasie
>>> class B(metaclass=A):      # Najpierw wykorzystywane są
        deskryptory danych w metaklasie
        data = 'mielonka'
>>> B[0]                      # Nazwy instancji metaklasy: widoczne
        tylko dla klasy
        's'
>>> B.__getitem__
<bound method A.__getitem__ of <class '__main__.B'>>
>>> I = B()
>>> I.data, B.data            # Zwykłe dziedziczone nazwy: widoczne
        dla instancji i klasy
        ('mielonka', 'mielonka')
>>> I[0]
TypeError: 'B' object does not support indexing
```

Można również zdefiniować metodę `__getattr__` w metaklasie, jednak można jej używać tylko do przetwarzania instancji jej *klas*, a nie zwykłych instancji. Jak zwykle, metoda nie jest przejmowana przez instancje klasy:

```
>>> class A(type):
    def __getattr__(cls, name):      # Przejęta przez metodę
        getitem klasy B
        return getattr(cls.data, name) # Nie można jej jednak
        uruchomić za pomocą wbudowanej operacji
>>> class B(metaclass=A):
    data = 'spam'
>>> B.upper()
'SPAM'
>>> B.upper
```

```
<built-in method upper of str object at 0x029E7420>
>>> B.__getattr__
<bound method A.__getattr__ of <class '__main__.B'>>
>>> I = B()
>>> I.upper
AttributeError: 'B' object has no attribute 'upper'
>>> I.__getattr__
AttributeError: 'B' object has no attribute '__getattr__'
```

Przeniesienie metody `__getattr__` do metaklasy nie pozwoli ominąć mankamentów przechwytywania wbudowanych operacji. W poniższym dalszym ciągu kodu do metody `__getattr__` metaklasy są kierowane jawne atrybuty, ale nie wbudowane operacje, pomimo że indeksowanie jest kierowane do metody `__getitem__` metaklasy z pierwszego przykładu opisanego w tym podrozdziale. Ewentualnie sugeruje to, że metoda `__getattr__` klasy w nowym stylu jest *specjalnym przypadkiem specjalnego przypadku*, przez co tym bardziej zalecane jest uproszczenie kodu, aby uniknąć zależności od tego rodzaju przypadków granicznych:

```
>>> B.data = [1, 2, 3]
>>> B.append(4)          # Jawne zwykłe nazwy są kierowane do metody getattr
metaklasy
>>> B.data
[1, 2, 3, 4]
>>> B.__getitem__(0)     # Jawne specjalne nazwy są kierowane do metody
getattr metaklasy
1
>>> B[0]                 # Natomiast wbudowana operacja metaklasy również
pomija metodę getattr metaklasy?
TypeError: 'A' object does not support indexing
```

Jak zatem widać, metaklasy są ciekawym tematem do badań, ale łatwo jest utracić ich ogólny obraz. Ze względu na ograniczone miejsce zostały tu pominięte drobne punkty. W tym rozdziale ważne jest przede wszystkim pokazanie, dlaczego warto korzystać z tego rodzaju narzędzia. Przejedźmy teraz do większych przykładów demonstrujących metaklasy w akcji. Jak się przekonamy, przeznaczeniem metaklas, podobnie jak wielu innych narzędzi w Pythonie, jest przede wszystkim ułatwianie utrzymywania kodu poprzez eliminację nadmiarowości.

Przykład — dodawanie metod do klas

W tym i kolejnym podrozdziale przestudiujemy przykłady dwóch częstych przypadków użycia metaklas — dodawania metod do klasy i automatycznego dekorowania wszystkich metod. Są to tylko dwie z wielu ról metaklas, które niestety pochłoną całe miejsce, jakie zostało nam w niniejszym rozdziale. Bardziej zaawansowane zastosowania metaklas można znaleźć w internecie. Przykłady te są jednak reprezentatywne dla działania metaklas i wystarczą do ilustrowania podstaw.

Co więcej, oba umożliwiają nam skontrastowanie dekoratorów klas i metaklas — nasz pierwszy przykład porównuje implementacje rozszerzania klas i opakowywania instancji oparte na metaklasach oraz dekoratorach, natomiast drugi najpierw zastosuje dekorator za pomocą metaklasy, a następnie za pomocą innego dekoratora. Jak zobaczymy, te dwa narzędzia są często wymienne, a nawet się dopełniają.

Ręczne rozszerzanie

We wcześniejszej części niniejszego rozdziału przyjrzaliśmy się szkieletowi kodu rozszerzającego klasy poprzez dodawanie do nich metod na różne sposoby. Jak widzieliśmy, proste dziedziczenie oparte na klasach wystarczy, jeśli dodatkowe metody są znane statycznie w momencie tworzenia klasy. Składanie za pomocą osadzania obiektów może często dać ten sam efekt. W bardziej dynamicznych scenariuszach wymagane są czasami inne techniki. Wystarczające mogą być zazwyczaj funkcje pomocnicze, jednak metaklasy udostępniają jawną strukturę i minimalizują koszty utrzymywania zmian w przyszłości.

Wcielmy zatem te pomysły w życie za pomocą działającego kodu. Rozważmy następujący przykład ręcznego rozszerzania klas. Dodaje on dwie metody do dwóch klas po ich utworzeniu:

```
# Ręczne rozszerzanie – dodawanie nowych metod do klas

class Client1:

    def __init__(self, value):
        self.value = value

    def spam(self):
        return self.value * 2

class Client2:

    value = 'ni?'

    def eggsfunc(obj):
        return obj.value * 4

    def hamfunc(obj, value):
        return value + 'szynka'

Client1.eggs = eggsfunc
Client1.ham = hamfunc
Client2.eggs = eggsfunc
Client2.ham = hamfunc

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bekon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bekon'))
```

Powyższy kod działa, ponieważ metody mogą zawsze być przypisywane do klasy po jej utworzeniu, dopóki przypisywane metody są funkcjami z dodatkowym pierwszym argumentem otrzymującym instancję `self`. Argument ten można wykorzystać do uzyskania dostępu do informacji o stanie dostępnych z instancji klasy, nawet jeśli funkcja definiowana jest niezależnie od klasy.

Po wykonaniu powyższego kodu otrzymujemy wynik metody zapisanej wewnątrz pierwszej klasy, a także dwóch metod dodanych do klas po fakcie:

```
c:\code> py -3 extend-manual.py
```

```
Ni!Ni!  
Ni!Ni!Ni!Ni!  
bekonszynka  
ni?ni?ni?ni?  
bekonszynka
```

Takie rozwiązanie działa dobrze w wyizolowanych przypadkach i można je wykorzystać do uzupełnienia klasy w dowolny sposób w czasie wykonywania. Ma ono jednak potencjalnie istotną wadę — musimy powtarzać kod rozszerzający dla każdej klasy potrzebującej tych metod. W naszym przypadku dodanie dwóch metod do dwóch klas nie było zbyt pracochłonne, jednak w bardziej skomplikowanych scenariuszach takie rozwiązanie może pochłaniać sporo czasu i być podatne na błędy. Jeśli kiedykolwiek zapomnimy o robieniu tego w sposób spójny lub będziemy musieli zmienić rozszerzenie, możemy napotkać problemy.

Rozszerzanie oparte na metaklasie

Choć rozszerzanie ręczne działa, w większych programach lepiej byłoby, gdybyśmy mogli automatycznie zastosować takie zmiany do całego zbioru klas. W ten sposób unikamy możliwości zepsucia rozszerzenia dla dowolnej klasy. Co więcej, zapisanie rozszerzenia w jednym miejscu lepiej obsługuje przyszłe zmiany — wszystkie klasy zbioru automatycznie pobiorą modyfikacje.

Jednym ze sposobów spełnienia tego celu jest użycie metaklas. Jeśli zapiszemy rozszerzenie w metaklasie, każda klasa deklarująca tę metaklasę będzie rozszerzona w ten sam, poprawny sposób i automatycznie pobierze wszelkie zmiany dokonane w przyszłości. Demonstруje to poniższy kod:

```
# Rozszerzenie za pomocą metaklasy – lepiej obsługuje przyszłe zmiany
def eggsfunc(obj):
    return obj.value * 4
def hamfunc(obj, value):
    return value + 'szynka'
class Extender(type):
    def __new__(meta, classname, supers, classdict):
        classdict['eggs'] = eggsfunc
        classdict['ham'] = hamfunc
        return type.__new__(meta, classname, supers, classdict)
class Client1(metaclass=Extender):
```

```

def __init__(self, value):
    self.value = value
def spam(self):
    return self.value * 2
class Client2(metaclass=Extender):
    value = 'ni?'
X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bekon'))
Y = Client2()
print(Y.eggs())
print(Y.ham('bekon'))

```

Tym razem obie klasy klienta rozszerzane są za pomocą nowych metod, ponieważ są instancjami metaklasy dokonującej rozszerzenia. Po wykonaniu powyższego kodu wynik będzie taki sam jak wcześniej — nie zmienialiśmy tego, co robi kod, po prostu dokonaliśmy refaktoryzacji, tak by rozszerzenie było wykonane w sposób bardziej czysty:

```
c:\code> py -3 extend-meta.py
```

```

Ni!Ni!
Ni!Ni!Ni!Ni!
bekonszynka
ni?ni?ni?ni?
bekonszynka

```

Warto zwrócić uwagę na to, że w tym przykładzie metaklasa nadal wykonuje stosunkowo statyczne zadanie — dodaje dwie znane metody do każdej klasy ją deklarującej. Tak naprawdę, gdyby jedynym naszym zadaniem było zawsze dodawanie tych samych dwóch metod do zbioru klas, również dobrze moglibyśmy zapisać je w normalnej klasie nadzędnej i dziedziczyć w klasach podrzędnych. W praktyce jednak struktura metaklasy obsługuje także o wiele bardziej dynamiczne zachowania. Przykładowo podmiotową klasę można także skonfigurować w oparciu o dowolną logikę w czasie wykonywania:

```

# Można również skonfigurować klasę w oparciu o testy w czasie wykonywania
class MetaExtend(type):
    def __new__(meta, classname, supers, classdict):
        if sometest():
            classdict['eggs'] = eggsfunc1
        else:
            classdict['eggs'] = eggsfunc2
        if someothertest():

```

```

        classdict['ham'] = hamfunc
    else:
        classdict['ham'] = lambda *args: 'Nie jest obsługiwane'
    return type.__new__(meta, classname, supers, classdict)

```

Metaklasy a dekoratory klas – runda 2.

Warto sobie przypomnieć, że dekoratory klas z poprzedniego rozdziału często pokrywają się z metaklasami z tego rozdziału w zakresie funkcjonalności. Wynika to z faktu, że:

- *Dekoratory klas* dowiązują ponownie nazwy klas do wyniku funkcji na końcu instrukcji `class`.
- *Metaklasy* działają dzięki przekierowaniu tworzenia obiektu klasy przez obiekt na końcu instrukcji `class`.

Choć są to nieco odmienne modele, w praktyce mogą zazwyczaj spełniać te same cele, choć w różny sposób. Jak widzieliśmy, dekorator klasy odpowiada bezpośrednio metodzie `__init__` metaklasy wywoływanej w celu zainicjowania nowo utworzonej klasy. Nie ma natomiast bezpośredniej analogii do metody `__new__` metaklasy (wywoywanej przede wszystkim w celu utworzenia klasy) ani do metody metaklasy (wykorzystywanej do przetwarzania instancji klasy). Jednak w wielu, a nawet większości przypadków użycia tych narzędzi nie jest wymagane wykonywanie wyżej opisanych dodatkowych kroków.

Tak naprawdę dekoratory klas można wykorzystać do zarządzania zarówno instancjami klasy, jak i samą klasą. W praktyce jednak metaklasy wprowadzają dodatkowe kroki w zarządzaniu instancjami, a dekoratory w tworzeniu klas. Zatem choć role metaklas i dekoratorów często się pokrywają, metaklasy najlepiej przydają się do zarządzania obiektemi klas. Teraz przełożymy te idee na kod.

Rozszerzenie oparte na dekoratorach

W przypadku czystego rozszerzania dekoratory często mogą zastępować metaklasy. Przykład z metaklasami z poprzedniego podrozdziału, dodający metody do klasy w czasie tworzenia, można także zapisać w postaci dekoratora klasy. W tym trybie dekoratory mniej więcej odpowiadają metodzie `__init__` metaklas, ponieważ obiekt klasy został już utworzony, zanim wywołyany zostaje dekorator. Podobnie jak w przypadku metaklas typ oryginalnej klasy zostaje zachowany, ponieważ nie zostaje wstawiona żadna warstwa obiektu opakowującego. Wynik poniższego kodu, dostępnego w pliku `extend-deco.py`, będzie taki sam jak poprzedniego kodu z metaklasą:

```

# Rozszerzenie za pomocą metaklasy – to samo co udostępnienie __init__ w
# metaklasie

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'szynka'

def Extender(aClass):
    aClass_eggs = eggsfunc
    # Zarządza klasą, a nie
    # instancją

```

```

aClass.ham = hamfunc                                # Odpowiednik __init__
metaklasy

    return aClass

@Extender

class Client1:                                     # Client1 =
Extender(Client1)

    def __init__(self, value):                      # Dowiązane ponownie na
        końcu instrukcji class

        self.value = value

    def spam(self):
        return self.value * 2

@Extender

class Client2:
    value = 'ni?'
X = Client1('Ni!')                                 # X jest instancją Client1
print(X.spam())
print(X.eggs())
print(X.ham('bekon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bekon'))

```

Innymi słowy, przynajmniej w pewnych przypadkach dekoratory potrafią zarządzać klasami równie łatwo, co metaklasy. Odwrotna zależność nie jest jednak tak oczywista — metaklasy można wykorzystać do zarządzania instancjami, jednak jedynie z użyciem pewnej dozy sztuczek. Zademonstrujemy to w kolejnym podrozdziale.

Zarządzanie instancjami zamiast klasami

Jak widzieliśmy przed chwilą, dekoratory klas mogą pełnić tę samą rolę w zakresie *zarządzania klasami*, co metaklasy. Metaklasy często spełniają także tę samą rolę, co dekoratory w zakresie *zarządzania instancjami*, jednak jest to nieco bardziej skomplikowane. A zatem:

- *Dekoratory klas* mogą zarządzać zarówno klasami, jak i instancjami.
- *Metaklasy* mogą zarządzać zarówno klasami, jak i instancjami, jednak instancje wymagają dodatkowej pracy.

To powiedziawszy, pewne aplikacje lepiej jest pisać w jeden lub drugi sposób. Rozważmy na przykład poniższy przypadek dekoratora klas z poprzedniego przykładu. Wykorzystywany jest on do wyświetlania komunikatu śledzenia za każdym razem, gdy dowolny, mający normalną nazwę atrybut instancji klasy jest pobierany:

```

# Dekorator klasy śledzi zewnętrzne pobrania atrybutów instancji
def Tracer(aClass):                                # Dla dekoratora @
    class Wrapper:

```

```

        def __init__(self, *args, **kargs):           # W momencie tworzenia
instancji

            self.wrapped = aClass(*args, **kargs)       # Użycie nazwy zakresu
obejmującego

        def __getattr__(self, attrname):
            print('Śledzenie:', attrname)             # Przechwytuje
wszystko poza .wrapped

            return getattr(self.wrapped, attrname)     # Deleguje do obiektu
wrapped

        return Wrapper

@Tracer

class Person:                                     # Person =
Tracer(Person)

    def __init__(self, name, hours, rate):         # Wrapper pamięta
Person

        self.name = name
        self.hours = hours
        self.rate = rate                           # Pobranie wewnętrz
metody nie jest śledzone

    def pay(self):
        return self.hours * self.rate

bob = Person('Robert', 40, 50)                   # bob to tak naprawdę
Wrapper

print(bob.name)                                    # Wrapper osadza Person
print(bob.pay())                                  # Wywołuje __getattr__

```

Po wykonaniu powyższego kodu dekorator wykorzystuje ponowne dowiązanie nazwy w celu opakowania obiektów instancji w obiekt, który tworzy wiersze śledzenia w poniższym wyniku:

```
c:\code> py -3 manage-inst-deco.py
Śledzenie: name
Robert
Śledzenie: pay
2000
```

Choć metaklasa może dać nam ten sam efekt, z konceptualnego punktu widzenia wydaje się to mniej oczywiste. Metaklasy zaprojektowane są specjalnie w celu zarządzania tworzeniem obiektów klas i mają interfejs przystosowany do tego celu. By wykorzystać metaklasę do zarządzania instancjami, musimy również wziąć odpowiedzialność za utworzenie klasy — jest to dodatkowy krok, pod warunkiem że utworzenie zwykłej klasy wystarczy. Poniższa metaklasa, dostępna w pliku *manage-inst-meta.py*, daje ten sam efekt i dane wyjściowe, co poprzedni dekorator:

```
# Zarządzanieinstancjami jak w poprzednim przykładzie, jednak za pomocą
metaklasy
```

```

def Tracer(classname, supers, classdict):
    # W momencie wywołania
    # tworzącego klasę

    aClass = type(classname, supers, classdict)           # Utworzenie klasy
    klienta

    class Wrapper:
        def __init__(self, *args, **kargs):               # W momencie tworzenia
            self.wrapped = aClass(*args, **kargs)

        def __getattr__(self, attrname):                  # Przechwytuje wszystko
            print('Śledzenie:', attrname)
            poza .wrapped

            return getattr(self.wrapped, attrname)         # Deleguje do obiektu
            wrapped

        return Wrapper

    class Person(metaclass=Tracer):                     # Tworzy Person za
        pomocą Tracer

        def __init__(self, name, hours, rate):          # # Wrapper pamięta Person
            self.name = name
            self.hours = hours
            self.rate = rate                            # Pobranie wewnętrz
            metody nie jest śledzone

        def pay(self):
            return self.hours * self.rate

    bob = Person('Robert', 40, 50)                      # bob to tak naprawdę
    Wrapper

    print(bob.name)                                     # Wrapper osadza Person
    print(bob.pay())                                    # Wywołuje __getattr__

```

Takie rozwiązanie działa, jednak opiera się na dwóch sztuczkach. Po pierwsze, musi wykorzystywać prostą funkcję zamiast klasy, ponieważ klasy podzielone `type` muszą się stosować do pewnych protokołów tworzenia obiektów. Po drugie, musi ręcznie tworzyć klasę podmiotową, wywołując `type`. Musi zwrócić opakowanie instancji, jednak metaklasy są także odpowiedzialne za tworzenie i zwracanie klasy podmiotowej. Tak naprawdę protokół metaklasy wykorzystujemy w tym przykładzie w taki sposób, by imitował on dekoratory, a nie odwrotnie. Ponieważ oba wykonywane są na końcu instrukcji `class`, w wielu rolach są one jedynie wariacjami na jeden temat. Powyższa wersja kodu z metaklasami daje po wykonaniu takie same dane wyjściowe, co dekorator:

```
c:\code> py -3 manage-inst-meta.py
```

```
Śledzenie: name
```

```
Robert
```

```
Śledzenie: pay
```

```
2000
```

Warto przestudiować obie wersje tego przykładu samodzielnie, by rozważyć ich wady i zalety. Generalnie jednak metaklasy z uwagi na swój projekt chyba najlepiej nadają się do zarządzania klasami. Dekoratory klas mogą zarządzać albo instancjami, albo klasami, choć mogą nie być najlepszą opcją w przypadku bardziej zaawansowanych ról pełnionych przez metaklasy, na które nie mamy w niniejszej książce miejsca. Więcej przykładów użycia metaklas jest dostępnych w internecie. Należy jednak pamiętać, że niektóre mogą być bardziej, a inne mniej poprawne (jak również ich autorzy mogą wiedzieć o Pythonie mniej niż Ty!).

Metaklasa równoważna dekoratorowi klasy?

W poprzednim podrozdziale pokazano, że metaklasy użyte do zarządzania instancjami wprowadzają dodatkowy krok podczas tworzenia klas i z tego powodu nie mogą w pełni zastępować dekoratorów we wszystkich sytuacjach. Ale czy możliwy jest odwrotny przypadek, tj. czy dekoratory mogą być zamiennikami klas?

Na wypadek, gdyby niniejszy rozdział nie spowodował jeszcze u czytających bólu głowy, rozważmy następujący alternatywny dla metaklas sposób kodowania: dekorator klasy zwracający instancję metaklasy:

```
# Dekorator może wywoływać metaklasę, ale nie odwrotnie bez type()

>>> class Metaclass(type):

    def __new__(meta, clsname, supers, attrdict):
        print('W M.__new__:')
        print([clsname, supers, list(attrdict.keys())])
        return type.__new__(meta, clsname, supers, attrdict)

>>> def decorator(cls):
    return Metaclass(cls.__name__, cls.__bases__, dict(cls.__dict__))

>>> class A:
    x = 1

>>> @decorator
    class B(A):
        y = 2

        def m(self): return self.x + self.y

W M.__new__:
['B', (<class '__main__.A'>,), ['__qualname__', '__doc__', 'm', 'y',
 '__module__']]

>>> B.x, B.y
(1, 2)

>>> I = B()
>>> I.x, I.y, I.m()
(1, 2, 3)
```

Powyższy kod niemal potwierdza równoważność obu narzędzi, ale w rzeczywistości tylko pod względem kierowania wywołań w chwili tworzenia klasy. Przypomnijmy, dekorator zasadniczo pełni tę samą rolę co metoda `__init__` metaklasy. Ponieważ w tym przykładzie dekorator

zwraca instancję metaklasy, zatem metaklasa — lub przynajmniej jej klasa nadrzędna `type` — wciąż jest tu obecna. Co więcej, ostatecznie dodatkowo po utworzeniu klasy wywoływana jest metaklasa, co nie jest idealnym rozwiązaniem w prawdziwym kodzie. Równie dobrze można przenieść metaklasę do pierwszego kroku tworzenia klasy:

```
>>> class B(A, metaclass=MetaClass): ... # Ten sam efekt, ale tworzona jest
    tylko jedna klasa
```

Wciąż jednak istnieje tu nadmiarowość narzędzi, a role dekoratora i metaklasy pokrywają się w praktyce. Choć dekoratory nie obsługują bezpośrednio omawianych wcześniej metod klasowych w metaklasach, metody i stan w obiektach pośredniczących utworzonych przez dekoratory pozwalają osiągnąć podobne efekty.

Odwrotna relacja raczej nie ma tu miejsca. Metaklasy generalnie nie da się przełożyć na dekoratora niemetaklasy, ponieważ klasa nie pojawi się, dopóki nie zakończy się wywołanie metaklasy. Jednak metaklasa *może* przyjąć postać prostego obiektu wywoływalnego, który wywołuje klasę `type` w celu bezpośredniego utworzenia klasy i przekazania jej do dekoratora. Innymi słowy, kluczowym punktem w tym modelu jest wywołanie klasy `type` w celu utworzenia klasy. Pod tym względem metaklasy i dekoratory klas są często funkcjonalnie równoważne z różnymi modelami *protoków kierowania*:

```
>>> def Metaclass(clsname, supers, attrdict):
        return decorator(type(clsname, supers, attrdict))
>>> def decorator(cls): ...
>>> class B(A, metaclass=MetaClass): ...           # Metaklasa może wywoływać
    dekorator i odwrotnie
```

W rzeczywistości metaklasa nie musi nawet zwracać instancji klasy `type` — może to być dowolny obiekt kompatybilny z oczekiwaniemi kodera klasy, co jeszcze bardziej zaciera różnicę pomiędzy dekoratorem a metaklasą:

```
>>> def func(name, supers, attrs):
        return 'mielonka'
>>> class C(metaclass=func):                   # Klasa, której metaklasa tworzy ciąg
    znaków!
        attr = 'huh?'
>>> C, C.upper()
('mielonka', 'MIELONKA')
>>> def func(cls):
        return 'spam'
>>> @func
class C:                                     # Klasa, której dekorator tworzy ciąg
    znaków!
        attr = 'huh?'
>>> C, C.upper()
('mielonka', 'MIELONKA')
```

Jak wspomniano wcześniej, role metaklasy i dekoratora, oprócz opisanych tu sztuczek, często określa moment użycia:

- Ponieważ *dekorator* jest uruchamiany po utworzeniu klasy, wprowadza dodatkowy krok w roli tworzenia klasy w trakcie *wykonywania* kodu.
- Ponieważ *metaklasa* musi tworzyć klasę, wprowadza dodatkowy krok w roli zarządzania instancją w trakcie *kodowania*.

Podsumowując, jedno narzędzie nie zastępuje drugiego. Metaklasy mogą tworzyć funkcjonalny nadzbiór, ponieważ mogą wywoływać dekoratory podczas tworzenia klas. Jednak są one trudniejsze w zrozumieniu i kodowaniu, a wiele ich ról całkowicie pokrywa się z dekoratorami. W praktyce potrzeba całkowitego tworzenia klasy jest zazwyczaj mniej ważna niż wykorzystanie procesu w ogólności.

Zamiast jednak dalej drążyć ten temat, zbadajmy role metaklas, co będzie bardziej typowym i praktycznym zajęciem. W kolejnym, kończącym niniejszy rozdział podrozdziale omówimy jeszcze jeden częsty przypadek użycia — automatyczne zastosowanie operacji do metod klas.

Przykład — zastosowanie dekoratorów do metod

Jak widzieliśmy w poprzednim podrozdziale, ponieważ metaklasy oraz dekoratory wykonywane są na końcu instrukcji `class`, często można je wykorzystywać *zamiennie*, choć z użyciem innej składni. Wybór pomiędzy tymi dwoma rozwiązaniami jest w wielu kontekstach dowolny. Można także wykorzystywać oba narzędzia *razem*, jako wzajemne dopełnienie. W niniejszym podrozdziale omówimy przykład tego typu połączenia — zastosowanie dekoratora funkcji do wszystkich metod klasy.

Ręczne śledzenie za pomocą dekoracji

W poprzednim rozdziale utworzyliśmy dwa dekoratory funkcji — jeden śledzący i zliczający wszystkie wywołania wykonywane do dekorowanej funkcji i drugi do obliczania czasu takich wywołań. Przybierały one różne formy; część z nich miała zastosowanie zarówno do funkcji, jak i metod, inne nie. Poniższa wersja zbiera ostateczne formy obu dekoratorów w plik modułu, tak by można było je wykorzystać ponownie i tu się do nich odwołać.

```
# Plik mytools.py – wybrane narzędzia dekoratorów

import time

def tracer(func):                                # Użycie funkcji, a nie klasy z
    __call__                                     # Inaczej self będzie jedynie
    calls = 0
    instancją dekoratora                         # Inaczej self będzie jedynie
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('wywołanie %s %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall
```

```

def timer(label='', trace=True):          # Dla argumentów dekoratora –
    zachowanie argumentów

    def onDecorator(func):                # Dla @ – zachowanie udekorowanej
        funkcji

            def onCall(*args, **kargs):      # W momencie wywołania –
                wywołanie oryginału

                    start = time.clock()       # Stan to zakresy + atrybuty
                funkcji

                    result = func(*args, **kargs)
                    elapsed = time.clock() - start
                    onCall.alltime += elapsed
                    if trace:
                        format = '%s%s: %.5f, %.5f'
                        values = (label, func.__name__, elapsed, onCall.alltime)
                        print(format % values)

                    return result
                onCall.alltime = 0
                return onCall
            return onCall
        return onDecorator

```

Jak wiemy z poprzedniego rozdziału, by użyć dekoratorów w sposób ręczny, po prostu importujemy je z modułu i piszemy kod ze składnią @ dekoratora przed każdą metodą, którą chcemy śledzić czy zmierzyć:

```

from decotools import tracer

class Person:
    @tracer
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):           # giveRaise =
tracer(giveRaise)
        self.pay *= (1.0 + percent)         # onCall pamięta giveRaise

    @tracer
    def lastName(self):                  # lastName = tracer(lastName)
        return self.name.split()[-1]

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)

```

```

print(bob.name, anna.name)
anna.giveRaise(.10)                                # Wykonuje onCall(anna, .10)
print(anna.pay)
print(bob.lastName(), anna.lastName())            # Wykonuje onCall(bob),
pamięta lastName

```

Po wykonaniu powyższego kodu otrzymujemy następujące dane wyjściowe. Wywołania udekorowanych metod przekierowywane są do logiki przechwytyjącej, a następnie delegującej wywołanie, ponieważ oryginalne nazwy metod zostały dowiązane do dekoratora:

```

c:\code> py -3 decoall-manual.py
wywołanie 1 __init__
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona

```

Śledzenie za pomocą metaklas oraz dekoratorów

Ręczny schemat dekoracji z poprzedniego podrozdziału działa, jednak wymaga dodania składni dekoracji przed każdą metodą, którą chcemy śledzić, a następnie późniejszego usunięcia tej składni, kiedy śledzenie nie będzie już pożądane. Jeśli chcemy śledzić każdą metodę klasy, w większych programach może się to stać dość żmudne. W bardziej dynamicznych kontekstach, w których rozszerzenia zależą od parametrów środowiska wykonawczego, może to nie być możliwe. Byłoby lepiej, gdybyśmy mogli w jakiś sposób automatycznie zastosować dekorator `tracer` do wszystkich metod klasy.

W przypadku metaklas możemy to zrobić — ponieważ są one wykonywane, kiedy klasa jest tworzona, są naturalnym miejscem na dodanie opakowań dekorujących do metod klas. Przeglądając słownik atrybutów klasy i sprawdzając istnienie w nim obiektów funkcji, możemy automatycznie wykonać metody za pośrednictwem dekoratora i ponownie dowiązać oryginalne nazwy do wyników. Efekt będzie taki sam, jak w przypadku automatycznego ponownego dowiązania nazw metod za pomocą dekoratorów, jednak możemy go zastosować w bardziej globalny sposób:

```

# Metaklasa dodająca dekorator śledzący do każdej metody klasy klienta
from types import FunctionType
from decotools import tracer
class MetaTrace(type):
    def __new__(meta, classname, supers, classdict):
        for attr, attrval in classdict.items():
            if type(attrval) is FunctionType:                      # Metoda?

```

```

        classdict[attr] = tracer(attrval)                      #
Udekrowanie jej

        return type.__new__(meta, classname, supers, classdict)      #
Utworzenie klasy

class Person(metaclass=MetaTrace):
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
print(bob.name, anna.name)
anna.giveRaise(.10)
print(anna.pay)
print(bob.lastName(), anna.lastName())

```

Po wykonaniu powyższego kodu wyniki są takie jak poprzednio. Wywołania metod przekierowywane są najpierw do dekoratora śledzącego, a później propagowane do oryginalnej metody:

```

c:\code> py -3 decoall-meta.py
wywołanie 1 __init__
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona

```

Widoczny powyżej wynik jest *kombinacją* działania dekoratora i metaklasy. Metaklasa automatycznie stosuje dekorator funkcji do każdej metody klasy w czasie tworzenia, a dekorator funkcji automatycznie przechwytuje wywołania metod w celu wyświetlenia komunikatów śledzenia w danych wyjściowych. Ta kombinacja po prostu działa dzięki uniwersalności obu narzędzi.

Zastosowanie dowolnego dekoratora do metod

Poprzedni przykład z metaklasą działa jedynie dla jednego określonego dekoratora funkcji — śledzenia. Uogólnienie kodu w taki sposób, by można było zastosować dowolny dekorator do wszystkich metod klasy, jest jednak trywialne. Wystarczy dodać zewnętrzną warstwę zakresu, by zachować pożądany dekorator, ponownie jak robiliśmy to w przypadku dekoratorów z poprzedniego rozdziału. W poniższym kodzie zapisano na przykład takie uogólnienie, a następnie wykorzystano je w celu ponownego zastosowania dekoratora śledzącego:

```
# Fabryka metaklas – zastosowanie dowolnego dekoratora do wszystkich metod
# klasy

from types import FunctionType
from deco.tools import tracer, timer

def decorateAll(decorator):

    class MetaDecorate(type):
        def __new__(meta, classname, supers, classdict):
            for attr, attrval in classdict.items():
                if type(attrval) is FunctionType:
                    classdict[attr] = decorator(attrval)
            return type.__new__(meta, classname, supers, classdict)

    return MetaDecorate

class Person(metaclass=decorateAll(tracer)):          # Zastosowanie dekoratora
do wszystkich metod

    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

    def lastName(self):
        return self.name.split()[-1]

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
print(bob.name, anna.name)
anna.giveRaise(.10)
print(anna.pay)
print(bob.lastName(), anna.lastName())
```

Po wykonaniu kodu w takiej postaci wynik będzie taki sam jak w poprzednich przykładach — nadal dekorujemy każdą metodę klasy klienta dekoratorem funkcji `tracer`, jednak robimy to w sposób bardziej ogólny:

```
c:\code> py -3 decoall-meta-any.py
wywołanie 1 __init__
```

```
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona
```

By teraz zastosować *inny* dekorator do metod, możemy po prostu zastąpić nazwę dekoratora w wierszu nagłówka klasy. By użyć na przykład pokazanego wyżej dekoratora funkcji `timer`, moglibyśmy wykorzystać dowolny z dwóch ostatnich wierszy nagłówka w poniższym kodzie, definiując naszą klasę — pierwszy przyjmuje domyślne argumenty dekoratora `timer`, natomiast drugi określa tekst podpisu:

```
class Person(metaclass=decorateAll(tracer)):      # Zastosowanie dekoratora
    tracer

class Person(metaclass=decorateAll(timer())):      # Zastosowanie dekoratora
    timer, argumenty domyślne

class Person(metaclass=decorateAll(timer(label='**'))): # Argumenty
    dekoratora
```

Warto zauważyć, że to rozwiązanie nie może obsługiwać niedomyślnych argumentów dekoratora różniących się dla poszczególnych metod, jednak można przekazać argumenty dekoratora mające zastosowanie dla wszystkich metod, jak wyżej. By to przetestować, wykorzystamy ostatnią z powyższych deklaracji metaklas w celu zastosowania dekoratora zliczającego czas i dodamy poniższe wiersze na końcu skryptu. Zobaczmy dodatkowe atrybuty informacyjne funkcji `timer`:

```
# Przy użyciu dekoratora timer – całkowity czas dla metody
print('*'*40)
print('%.5f' % Person.__init__.alltime)
print('%.5f' % Person.giveRaise.alltime)
print('%.5f' % Person.lastName.alltime)
```

Nowy wynik będzie następujący. Metaklasa opakowuje teraz metody w dekoratory zliczające czas, dzięki czemu możemy dla każdej metody klasy zobaczyć, ile trwa każde wywołanie:

```
c:\code> py -3 decoall-meta-any2.py
**__init__: 0.00001, 0.00001
**__init__: 0.00001, 0.00002
Robert Zielony Anna Czerwona
**giveRaise: 0.00001, 0.00001
110000.0
**lastName: 0.00001, 0.00001
**lastName: 0.00001, 0.00002
Zielony Czerwona
```

```
0.00002  
0.00001  
0.00002
```

Metaklasy a dekoratory klas — runda 3. (i ostatnia)

Dekoratory klas również w tym punkcie pokrywają się w metaklasami. Poniższa wersja kodu zastępuje metaklasę z poprzedniego przykładu dekoratorem klasy. Definiuje i wykorzystuje *dekorator klasy, który zastosuje dekorator funkcji* do wszystkich metod klasy. Choć poprzednie zdanie może brzmieć bardziej jak instrukcja zen niż techniczny opis, wszystko to działa dość naturalnie — dekoratory Pythona obsługują dowolne zagnieżdżanie i kombinacje:

```
# Fabryka dekoratora klas – zastosowanie dowolnego dekoratora do wszystkich  
# metod klasy  
  
from types import FunctionType  
  
from decoTools import tracer, timer  
  
def decorateAll(decorator):  
  
    def DecoDecorate(aClass):  
  
        for attr, attrval in aClass.__dict__.items():  
  
            if type(attrval) is FunctionType:  
  
                setattr(aClass, attr, decorator(attrval))      # Nie __dict__  
  
    return aClass  
  
    return DecoDecorate  
  
@decorateAll(tracer)                                # Użycie  
dekoratora klasy  
  
class Person:                                         # Stosuje  
dekorator funkcji do metod  
  
    def __init__(self, name, pay):                      # Person =  
decoratAll(..)(Person)  
  
        self.name = name                               # Person =  
DecoDecorate(Person)  
  
        self.pay = pay  
  
    def giveRaise(self, percent):  
        self.pay *= (1.0 + percent)  
  
    def lastName(self):  
  
        return self.name.split()[-1]  
  
bob = Person('Robert Zielony', 50000)  
anna = Person('Anna Czerwona', 100000)  
print(bob.name, anna.name)
```

```

anna.giveRaise(.10)
print(anna.pay)
print(bob.lastName(), anna.lastName())

```

Po wykonaniu kodu w powyższej postaci dekorator klasy zastosuje dekorator funkcji `tracer` do każdej metody i przy wywołaniu zwróci komunikaty śledzenia (wynik będzie taki sam jak w poprzedniej wersji tego przykładu z metaklasą):

```

c:\code> py -3 decoall-deco-any.py
wywołanie 1 __init__
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona

```

Warto zauważyć, że dekorator klasy zwraca oryginalną, rozszerzoną klasę, a nie jej warstwę opakowującą (co często się zdarza, kiedy zamiast tego opakowuje się obiekty instancji). W przypadku wersji z metaklasą zachowujemy typ oryginalnej klasy — instancja `Person` jest instancją klasy `Person`, a nie jakiejś klasy opakowującej. Tak naprawdę ten dekorator klasy zajmuje się jedynie tworzeniem klasy. Wywołania tworzące instancje nie są w ogóle przechwytywane.

To rozróżnienie ma znaczenie w programach, które wymagają sprawdzania typów dla instancji w celu zwrócenia oryginalnej klasy, a nie opakowania. Przy rozszerzaniu klasy zamiast instancji dekoratory klas mogą zachować typ oryginalnej klasy. Metody klas nie są ich oryginalnymi funkcjami, ponieważ są ponownie dowiązywane do dekoratorów, jednak w praktyce jest to mniej istotne; w rozwiązaniu z metaklasą jest zresztą tak samo.

Warto również zauważyć, że tak jak w wersji z metaklasą struktura ta nie może obsługiwać argumentów dekoratorów funkcji różniących się dla poszczególnych metod, jednak jest w stanie obsługiwać takie argumenty, jeśli zostaną one zastosowane do wszystkich metod. By na przykład użyć tego schematu do zastosowania dekoratora zliczającego czas, wystarczający będzie jeden z dwóch ostatnich wierszy zaprezentowanych poniżej, wstawiony tuż przed definicją klasy — pierwszy z nich wykorzystuje domyślne argumenty dekoratora, natomiast drugi przekazuje argument w sposób jawnny:

```

@decorateAll(tracer)           # Udekorowanie wszystkiego za pomocą
dekoratora tracer

@decorateAll(timer())          # Udekorowanie wszystkiego za pomocą
dekoratora timer, argumenty domyślne

@decorateAll(timer(label='@')) # To samo, ale z przekazanym argumentem
dekoratora

```

Tak jak wcześniej użyjemy ostatniego wiersza z dekoratorem i dodamy następujący kod na końcu skryptu w celu przetestowania naszego przykładu z innym dekoratorem (oczywiście istnieją lepsze schematy zarówno testowania, jak i mierzenia czasu, ale jesteśmy na końcu rozdziału; kod zawsze można ulepszyć odpowiednio do potrzeb):

```
# Przy użyciu dekoratora timer – całkowity czas dla metody
```

```

print('-*40)
print('%.5f' % Person.__init__.alltime)
print('%.5f' % Person.giveRaise.alltime)
print('%.5f' % Person.lastName.alltime)

```

Widoczny będzie ten sam wynik — dla każdej metody zliczamy czas każdego z wywołań, jednak tym razem do dekoratora `timer` przekazaliśmy inny argument `label`:

```

c:\code> py -3 decoall-deco-any2.py
@@__init__: 0.00001, 0.00001
@@__init__: 0.00001, 0.00002
Robert Zielony Anna Czerwona
@@giveRaise: 0.00001, 0.00001
110000.0
@@lastName: 0.00001, 0.00001
@@lastName: 0.00001, 0.00002
Zielony Czerwona
-----
0.00002
0.00001
0.00002

```

I wreszcie można łączyć dekoratory tak, aby każdy z nich był uruchamiany podczas wywoływania metody, ale zazwyczaj wymaga to wprowadzenia zmian do napisanego już kodu. Bezpośrednie zagnieźdzenie jednego dekoratora powoduje śledzenie lub mierzenie czasu tworzenia drugiego. Umieszczenie obu dekoratorów w osobnych wierszach powoduje śledzenie lub mierzenie czasu opakowania drugiego dekoratora przed uruchomieniem oryginalnej metody. Metaklasy też nie najlepiej sobie tutaj radzą:

```

@decorateAll(tracer(timer(label='@@')))    # Dekorator śledzący, stosujący
dekorator mierzący czas

class Person:

@decorateAll(tracer)                      # Śledzenie opakowania metody
onCall, mierzenie czasu

# działania metod

@decorateAll(timer(label='@@'))

class Person:

@decorateAll(timer(label='@@'))

@decorateAll(tracer)                      # Mierzenie czasu opakowania metody
onCall, śledzenie metod

class Person:

```

Dalsze rozważania na ten temat musimy zostawić jako proponowane ćwiczenie, ponieważ tutaj nie ma już na nie czasu ani miejsca.

Jak widać, metaklasy i dekoratory klas są nie tylko wymienne, ale często się także dopełniają. Oba rozwiązania udostępniają zaawansowane sposoby dostosowywania obiektów i klas do własnych potrzeb, a także zarządzania nimi, ponieważ pozwalają na wstawianie kodu do procesu tworzenia klas. Choć niektóre bardziej zaawansowane zastosowania mogą być zapisane w kodzie z użyciem jednego bądź drugiego rozwiązania, wybór lub połączenie tych dwóch narzędzi w dużej mierze pozostaje w gestii programisty.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy metaklasy, a także przyjrzaliśmy się przykładom ich użycia. Metaklasy umożliwiają nam wejście w protokół tworzenia klas w Pythonie w celu zarządzania klasami zdefiniowanymi przez użytkownika lub rozszerzenia ich. Ponieważ pozwalają one automatyzować ten proces, mogą być lepszym rozwiązaniem dla twórców API od kodu pisanego ręcznie czy funkcji pomocniczych. Ponieważ hermetyzują kod tego typu, mogą zminimalizować koszty utrzymywania kodu o wiele lepiej od innych rozwiązań.

Przy okazji widzieliśmy także, że role dekoratorów klas oraz metaklas często się pokrywają. Ponieważ oba narzędzia wykonywane są na końcu instrukcji `class`, czasami można ich używać wymiennie. Dekoratory klas można wykorzystać do zarządzania obiektami zarówno klas, jak i instancji. Metaklasy także to potrafią, choć są przeznaczone bardziej dla klas.

Ponieważ niniejszy rozdział omawiał zagadnienie zaawansowane, przejdziemy przez tylko kilka pytań quizu omawiających podstawy (każdy, kto w rozdziale o metaklasach dotarł aż tak daleko, zasługuje na dodatkową nagrodę!). A ponieważ jest to ostatnia część książki, opuścimy także ćwiczenia kończące tę część. Polecam zajrzenie teraz do dodatków zawierających m.in. zmiany wprowadzone w Pythonie i rozwiązania ćwiczeń z poprzednich części książki. W ostatnim dodatku znajdują się próbki typowych programów do samodzielnego przestudiowania.

Po skończeniu quizu oficjalnie dochodzimy do końca technicznego materiału. Następny i ostatni rozdział zawiera krótkie wnioski podsumowujące książkę.

Sprawdź swoją wiedzę — quiz

1. Czym jest metaklasa?
2. W jaki sposób deklaruje się metaklasę klasy?
3. W jaki sposób dekoratory klas pokrywają się z metaklasami w zakresie zarządzania klasami?
4. W jaki sposób dekoratory klas pokrywają się z metaklasami w zakresie zarządzania instancjami?
5. Co można zaliczyć do głównych metod programisty Pythona — dekoratory klas czy metaklasy? (Odpowiedź proszę podać zgodnie z popularnym skeczem Monty Pythona).

Sprawdź swoją wiedzę — odpowiedzi

1. Metaklasa to klasa wykorzystywana do utworzenia klasy. Normalne klasy domyślnie są instancjami klasy `type`. Metaklasy są zazwyczaj klasami podlegającymi `type`, redefiniującymi protokół tworzenia klas w celu dostosowania wywołania tworzącego klasę do własnych potrzeb na końcu instrukcji `class`. Zazwyczaj redefiniują one metody `__new__` oraz `__init__` w celu uzyskania dostępu do protokołu tworzenia klas. Metaklasy można także zapisywać w inny sposób — na przykład jako proste funkcje — jednak są one odpowiedzialne za tworzenie i zwracanie obiektów dla nowej klasy. Metaklasy mogą również zawierać metody i dane określające działanie ich klas. Stanowią drugą ścieżkę w przeszukiwaniu dziedziczenia. Jednak ich atrybuty są dostępne dla instancji ich klas, a nie dla instancji ich instancji.
2. W Pythonie 3.x należy wymienić pożądaną metaklasę jako argument ze słowem kluczowym w nagłówku instrukcji `class` — w postaci `class C(metaclass=M)`. W Pythonie 2.x należy zamiast tego użyć atrybutu klasy — formy `__metaclass__ = M`. W wersji 3.x wiersz nagłówka klasy może także wymieniać normalne klasy nadzędne przed argumentem ze słowem kluczowym `metaclass`. W wersji 2.x również można dziedziczyć klasę `object`, jednak zazwyczaj jest to opcjonalne rozwiązanie.
3. Ponieważ obie wersje są wywoływane automatycznie na końcu instrukcji `class`, dekoratory klas oraz metaklasy mogą być wykorzystane do zarządzania klasami. Dekoratory dowiązują ponownie nazwy klas do wyników obiektów wywoływalnych, natomiast metaklasy przekierowują tworzenie klasy do obiektu wywoływalnego, jednak oba punkty zaczepienia można wykorzystywać w podobnych celach. By zarządzać klasami, dekoratory po prostu rozszerzają i zwracają oryginalny obiekt klasy. Metaklasy rozszerzają klasę po utworzeniu jej. Jeżeli trzeba zdefiniować nową klasę, dekoratory w tej roli nie do końca się sprawdzają, ponieważ oryginalna klasa została już utworzona.
4. Ponieważ obie wersje wywoływane są automatycznie na końcu instrukcji `class`, dekoratory klas oraz metaklasy mogą być wykorzystane do zarządzania instancjami klas, wstawiając obiekt opakowujący, który przechwytuje wywołania tworzące instancje. Dekoratory mogą dowiązywać ponownie nazwę klasy do obiektu wywoływalnego wykonywanego w momencie tworzenia instancji, który zachowuje oryginalny obiekt klasy. Metaklasy mogą robić to samo, jednak muszą także tworzyć obiekt klasy, dlatego ich użycie jest w tej roli nieco bardziej skomplikowane.
5. Nasza główna metoda to dekoratory... dekoratory i metaklasy... metaklasy i dekoratory... Dwie metody: metaklasy i dekoratory... i bezwzględna skuteczność... Trzy metody: metaklasy, dekoratory i bezwzględna skuteczność... i niemalże fanatyczne oddanie Pythonowi... Cztery... Nie... Wśród naszych metod... Wśród naszych metod są takie elementy jak... metaklasy, dekoratory... Wejdziemy jeszcze raz...

[1] Przytoczę komunikat, z którym się spotkałem w wersji 3.3: *TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases* („Konflikt metaklas: metaklasa klasy pochodnej musi być (nieściśłą) podklasą metaklas wszystkich swoich klas bazowych”). Oznacza on błędne użycie modułu jako klasy nadzędnej, jednak metaklasy nie są tak opcjonalne, jak by się programistom wydawało — tym zagadnieniem zajmiemy się w podsumowaniu następnego rozdziału.

Rozdział 41. Wszystko, co najlepsze

Witaj w ostatnim rozdziale książki! Skoro zaszedłeś tak daleko, chciałbym na zakończenie, zanim wypłyniesz na szerokie wody programowania, powiedzieć Ci kilka słów na temat ewolucji Pythona. Będzie to oczywiście subiektywny punkt widzenia, niemniej jednak ważny dla wszystkich użytkowników tego języka.

Miałeś okazję ujrzeć Pythona w całej okazałości, w tym kilka zaawansowanych funkcjonalności, które mogą wydawać się sprzeczne z jego paradigmatem skryptowym. Choć zrozumiałym jest, że wielu programistów akceptuje taki stan rzeczy, to w otwartych projektach bardzo ważne jest również zadawanie pytań „dlaczego?”. Ostatecznie kierunek rozwoju Pythona — mówię to z całą powagą — zależy po części również od Ciebie.

Paradoks Pythona

Skoro przeczytałeś tę książkę lub jej większą część, powinieneś umieć oceniać kompromisy w Pythonie. Jak się przekonałeś, jest to potężny, ekspresyjny, a nawet przyjemny język programowania, który może być Twoim kluczem do wszelkiej technologii, jaką teraz wybierzesz. Jednocześnie dowiedziałeś się, że w dzisiejszym Pythonie tkwi swego rodzaju paradoks: język ten rozwinął się do tego stopnia, że zawiera narzędzia, które wielu programistów uważa za nadmiarowe i osobliwie zaawansowane. A tempo rozwoju wydaje się tylko zwiększać.

Jako jeden z pierwszych propagatorów Pythona obserwowałem jego metamorfozę, jaką na przestrzeni lat przeszedł od prostego do zaawansowanego narzędzia, którego zasięg dalej się rozszerza. Pod wieloma względami stał się przynajmniej tak zaawansowany jak inne języki, które doprowadziły wielu z nas do Pythona, i podobnie jak w przypadku innych języków, w nieunikniony sposób przyczynił się do rozwoju kultury, w której niejasność jest oznaką honoru.

Jest to tak sprzeczne z pierwotnymi celami Pythona, jak to tylko możliwe. Aby przekonać się, co mam na myśli, wpisz w interaktywnej sesji polecenie `import this` — kredo wielokrotnie cytowane w tej książce w kontekstach, w których zostało wyraźnie naruszone. Na wielu płaszczyznach podstawowe ideały jawności, prostoty i braku nadmiarowości zostały albo naiwnie zapomniane, albo beztrosko porzucone.

W efekcie powstał język i społeczność, które dzisiaj można byłoby opisać, używając niektórych terminów użytych w rozdziale 1. w notatce na temat języka Perl. Choć Python ma wiele do zaoferowania, ten trend grozi zanegowaniem wielu jego postrzeganych zalet, o czym mowa w następnym podrozdziale.

„Opcjonalne” cechy języka

Na początku poprzedniego rozdziału, aby podkreślić postrzeganą niejasność metaklas, przytoczyłem cytat, że nie są one interesujące dla 99% programistów. To stwierdzenie nie jest do końca ścisłe, nie tylko liczbowo. Autor cytatu jest znanym współtwórcą i entuzjastą Pythona

od jego najwcześniejszych dni i nie chcę nikogo niesprawiedliwie oceniać. Co więcej, tak naprawdę sam często stwierdzałem w kolejnych wydaniach tej książki, że tego rodzaju informacje o języku są niejasne.

Problem polega jednak na tym, że takie stwierdzenia w rzeczywistości dotyczą tylko osób, które pracują samotnie i zawsze używają kodu, który same napisały. Gdy tylko ktoś w organizacji skorzysta z „opcjonalnej” zaawansowanej funkcji języka, funkcjonalność ta przestaje być opcjonalna i zaczyna być skutecznie narzucana *wszystkim* osobom. To samo dotyczy zewnętrznego oprogramowania, którego używasz do tworzenia swoich systemów: jeśli autor korzysta z zaawansowanej lub obcej funkcji języka, nie jest to już dla Ciebie całkowicie opcjonalna funkcja, ponieważ trzeba ją zrozumieć, aby móc ją stosować i zmieniać w kodzie.

Powyższy wniosek dotyczy nie tylko zaawansowanych zagadnień opisanych w tej książce, w tym wymienionych na początku poprzedniego rozdziału jako „magiczne” punkty zaczepienia, ale też innych: generatorów, dekoratorów, slotów, właściwości, deskryptorów, metaklas, menedżerów kontekstu, domknięć, funkcji super, menedżerów przestrzeni nazw, standardu Unicode, adnotacji funkcji, względnych importów, argumentów opartych wyłącznie na słowach kluczowych, klas, statycznych metod, a nawet wyrażeń o niejasnych zastosowaniach i przeciążanych operatorów.

Programista i program wykorzystujący powyższe narzędzia automatycznie stają się częścią *wymaganej* przez Ciebie bazy wiedzy.

Aby przekonać się, jak bardzo może to być zniechęcające, wystarczy przeanalizować opisaną w rozdziale 40. *procedurę dziedziczenia klas w nowym stylu*. Jest to niesłychanie zagmatwany model, w którym do zrozumienia nawet tak prostego zagadnienia jak odwzorowywanie nazw wymagana jest wiedza o deskryptorach i metaklasach. Podobnym intelektualnym wyzwaniem jest rozdział 32. poświęcony nieprzyzwoicie niejawnemu i sztucznemu algorytmowi MRO, który musi znać programista czytający kod wykorzystujący to narzędzie.

Ostatecznym efektem takiego nadmiaru inżynierii jest wymóg radykalnego zwiększenia ilości wiedzy niezbędnej do przyswojenia, jak również pojawienie się rzeszy programistów, którzy tylko częściowo rozumieją używane przez siebie narzędzia. Jest to oczywiście sytuacja sprzeczna z ideą języka skryptowego i daleka od ideału dla tych osób, które chcą używać Pythona w prosty sposób.

Przeciwko niepokojącym usprawnieniom

Powyższe wnioski dotyczą również wielu nadmiarowych funkcjonalności, które poznaliśmy wcześniej, m.in. opisanej w rozdziale 7. metody `str.format` i instrukcji `with` z rozdziału 34. Są to narzędzia zapożyczone z innych języków, pokrywające się z innymi, istniejącymi w Pythonie od długiego czasu.

Powiedzmy uczciwie: w ciągu ostatnich lat w Pythonie zaroło się od nadmiarowych funkcjonalności. Jak proponowałem w przedmowie i jak się osobiście przekonałeś, dzisiejszy Python jest pełen duplikatów i rozszerzeń, których część — spośród wielu opisanych w tej książce — pokazuje tabela 41.1.

Tabela 41.1. Próbka nadmiarowości i eksplozji funkcjonalności w Pythonie

Kategoria	Opis
3 główne paradygmaty	Proceduralny, funkcjonalny i obiektowy
2 niekompatybilne wersje	2.x i 3.x z klasami w nowym stylu w obu wersjach
3 narzędzia formatujące ciągi znaków	Wyrażenie %, <code>str.format</code> , <code>string.Template</code>

4 narzędzia dostępu do atrybutów	Metody <code>__getattr__</code> , <code>__getattribute__</code> , właściwości, deskryptory
2 instrukcje finalizujące	Instrukcje <code>try/finally</code> , <code>with</code>
4 odmiany wyrażeń	Lista, generator, zbiór, słownik
3 narzędzia rozszerzające klasy	Funkcje, dekoratory, metaklasy
4 rodzaje metod	Metody instancji, klasy, metaklasy, metody statyczne
2 systemy zapisywania atrybutów	Słowniki, sloty
4 odmiany importu	Moduł, pakiet, względny import pakietu, pakiet przestrzeni nazw
2 protokoły kierowania klas nadzędnych	Bezpośrednie wywołania, funkcja <code>super</code> + algorytm MRO
5 form instrukcji przypisujących wartości	Podstawowa, wielonazwowa, rozszerzona, sekwencyjna, gwiazdżysta
2 typy funkcji	Zwykła, generator
5 form argumentów funkcji	Zwykły, nazwa=wartość, <code>*pargs</code> , <code>**kargs</code> , tylko słowa kluczowe
2 źródła działania klas	Klasy nadzędne, metaklasy
4 opcje zachowywania stanu	Klasy, domknięcia, atrybuty funkcji, klasy mutowalne
2 modele klas	Model typowy oraz model w nowym stylu w wersji 2.x, model wyłącznie w nowym stylu w wersji 3.x
2 modele Unicode	Opcjonalne w wersji 2.x, obowiązkowe w wersji 3.x
2 tryby PyDoc	Klient interfejsu graficznego wymagany we wszystkich przeglądarkach w wersji 3.x
2 schematy przechowywania kodu bajtowego	Oryginalny, <code>__pycache__</code> tylko w wersji 3.x

Jeśli zależy Ci na Pythonie, poświeć chwilę na przejrzenie powyższej tabeli. Obrazuje ona eksplozję funkcjonalności i liczby narzędzi — 59 pojęć stanowi porządną dawkę dla początkujących. Większość kategorii miała wspólny początek w Pythonie, wiele kolejnych rozwinęło się na wzór innych języków i tylko kilka ostatnich można uproszczyć pod pretekstem, że najnowszy Python jest jedynym Pythonem, który ma znaczenie dla jego użytkowników.

W tej książce starałem się unikać nieuzasadnionej złożoności, ale w praktyce zarówno zaawansowane, jak i nowe narzędzia zachęcają do tego, aby je przystosowywać do własnych celów, przy czym nierzadko jedynym powodem jest osobiste pragnienie programisty wykazania własnej biegłości. W rezultacie większość dzisiejszego kodu Pythona jest zaśmiecona tymi

skomplikowanymi, obcymi narzędziami. Oznacza to, że *nic nie jest naprawdę „opcjonalne”, skoro faktycznie nie jest opcjonalne*.

Złożoność a siła

Z tego powodu niektórzy weterani Pythona (w tym ja) czasami martwią się, że język ten w miarę upływu czasu staje się coraz większym i coraz bardziej skomplikowanym językiem. Nowe funkcjonalności wprowadzane przez weteranów, neofitów, a nawet amatorów podnoszą intelektualną poprzeczkę dla początkujących użytkowników. Fundamentalne idee Pythona, takie jak dynamiczne typowanie i wbudowane typy, pozostały raczej niezmienione, ale zaawansowane dodatki są lekturą obowiązkową dla każdego programisty. Z tego powodu postanowiłem je omówić w tej książce, mimo że pominąłem je w poprzednich wydaniach. Nie można zignorować zaawansowanych funkcjonalności, jeśli są stosowane w kodzie, który trzeba zrozumieć.

Z drugiej strony, jak wspomniałem w rozdziale 1., dla większości obserwatorów Python jest wciąż *prostszy* od większości współczesnych języków, a złożoność wynika z roli, jaką musi pełnić. Wzbogacił się wprawdzie o wiele takich samych narzędzi, jakie są obecne w Javie, C# i C++, ale mają one mniejszą wagę w kontekście dynamicznie pisanej języka skryptowego. Mimo całego swojego wieloletniego rozwoju Python wciąż jest językiem stosunkowo łatwym do nauczenia się i stosowania w porównaniu z innymi językami, a nowi użytkownicy mogą wybierać zaawansowane tematy, gdy pojawi się taka potrzeba.

Trzeba też przyznać, że programiści aplikacji większość czasu poświęcają *bibliotekom i rozszerzeniom*, a nie zaawansowanym i czasami tajemnym funkcjom języka. Na przykład książka *Programming Python* — kontynuacja niniejszej — dotyczy głównie powiązań Pythona z bibliotekami aplikacyjnymi przeznaczonymi do takich zastosowań, jak tworzenie interfejsów graficznych, obsługę bazy danych i komunikacji przez internet, a nie ezoterycznym narzędziom językowym (aczkolwiek standard Unicode wciąż narzuca się na wielu etapach, a po drodze nieustannie pojawia się dziwne wyrażenie generatora z instrukcją `yield`).

Co więcej, efektem ubocznym tego wzrostu jest większa *moc* Pythona. Jeżeli używa się go umiejętnie, narzędzia takie jak dekoratory i metaklasy nie tylko „są fajne”, ale pozwalają kreatywnym programistom tworzyć bardziej elastyczne i przydatne interfejsy na użytek innych programistów. Jak się przekonaliśmy, pomagają też rozwiązywać problemy związane z enkapsulacją i utrzymaniem kodu.

Prostota a elitarność

Pozostawiam Twojej opinii, czy powyższe wywody uzasadniają potencjalne poszerzenie wymaganej wiedzy o Pythonie. Lepiej lub gorzej, zazwyczaj decydują o tym umiejętności programisty. Bardziej zaawansowani użytkownicy lubią bardziej skomplikowane narzędzia i zwykle zapominają o ich odbiorze przez osoby z innego obozu. Na szczęście nie zawsze tak jest; dobrzy programiści rozumieją, że *dobra inżynieria charakteryzuje się prostotą*, a zaawansowanych narzędzi należy używać tylko wtedy, gdy jest to uzasadnione. Dotyczy to każdego języka programowania, w szczególności Pythona, który jest często prezentowany nowym lub początkującym programistom jako rozszerzalne narzędzie.

Jeśli nadal Cię to nie przekonuje, pamiętaj, że wielu użytkowników Pythona nie czuje się komfortowo nawet w *podstawowym programowaniu obiektowym*. Uwierz mi, że tak jest — spotkałem tysiące takich ludzi. Python nigdy nie był trywialnym tematem, ale raporty z programistycznych okopów są w tej kwestii jednoznaczne: nieuzasadniona, dodatkowa złożoność nigdy nie jest mile widzianą cechą, szczególnie gdy jej źródłem są osobiste preferencje nielicznych i nireprezentatywnych użytkowników. Niezależnie od tego, czy jest to ich zamierzane działanie, czy nie, jest zazwyczaj postrzegane jako *elitarność*, czyli nieproduktywny i arogancki sposób myślenia, dla którego nie ma miejsca w tak szeroko stosowanym narzędziu, jakim jest Python.

Jest to również problem społeczny, dotyczący zarówno programistów, jak i twórców języków. Jednak w „prawdziwym świecie”, w którym liczy się otwarte oprogramowanie, systemy oparte na Pythonie, wymagające od użytkowników opanowania niuansów metaklas, deskryptorów itp., powinny być skalowane odpowiednio do oczekiwania rynku.

Mam nadzieję, że książka ta spełni swoje zadanie i przekonasz się, że prostota programowania jest jednym z najważniejszych i najtrwalszych atutów języka.

Końcowe wnioski

Takie są więc obserwacje kogoś, kto od dwóch dekad używa, uczy i propaguje Pythona i życzy mu wszystkiego najlepszego na przyszłość. Oczywiście, żadne z powyższych przemyśleń nie jest nowe. Nawet rozwój tej książki na przestrzeni lat świadczy o własnym rozwoju Pythona, co można traktować jako *ironiczną pochwałę* jego pierwotnej koncepcji jako narzędzia, które upraszcza programowanie i jest dostępne zarówno dla ekspertów, jak i niespecjalistów. Sądząc po samej wadze języka, wydaje się, że marzenie to zostało zaniedbane, albo wręcz całkowicie porzucone.

Jednak popularność Pythona raczej nie wykazuje oznak spadku — jest to potężna przeciwwaga dla problemów związanych ze złożonością. Dzisiejszy świat może być — co zrozumiałe — bardziej zainteresowany stosowaniem Pythona w praktyce w takiej formie, w jakiej jest, niż jego pierwotnymi i nieco idealistycznymi celami. Za pomocą Pythona można wykonywać wiele skomplikowanych zadań programistycznych, a to jest wystarczający powód, aby polecać go do różnych zastosowań. Pomijając pierwotne cele, masowa popularność tego języka jest swego rodzaju sukcesem, jednak na ostateczny werdykt przyjdzie nam jeszcze poczekać.

Jeżeli wciąż interesuje Cię ewolucja i krzywa uczenia Pythona, zapoznaj się z obszernym artykułem *Answer Me These Questions Three...* („Odpowiedz mi tylko na trzy pytania”), który napisałem na ten temat w 2012 r., dostępnym pod adresem <http://learning-python.com/pyquestions3.html>. Znajdziesz tam odpowiedzi na ważne, pragmatyczne pytania, które są kluczowe dla przyszłości Pythona i zasługują na więcej uwagi niż poświęcona w tej książce. Są to kwestie bardzo subiektywne, tekst nie jest filozoficzny, a ta książka już przekroczyła dopuszczalną liczbę stron.

Co ważniejsze, w otwartym projekcie, jakim jest Python, odpowiedzi na tego rodzaju pytania muszą być stawiane na nowo przez każdą falę przybyszów. Mam nadzieję, że fala, której dajesz się ponosić, będzie wykazywała tyle samo zdrowego rozsądku, co zabawa w planowanie przyszłości Pythona.

Dokąd dalej?

To już jest koniec. Oficjalnie dotarliśmy do końca niniejszej książki. Po poznaniu Pythona na wylot kolejnym krokiem powinno być zapoznanie się z bibliotekami, technikami i narzędziami dostępnym w dziedzinie aplikacji, w której chcemy pracować.

Ponieważ Python jest tak szeroko wykorzystywany, można znaleźć mnóstwo zasobów dotyczących wykorzystywania go w prawie dowolnej dziedzinie, jaką można sobie wymyślić — od graficznych interfejsów użytkownika po internet, bazy danych, programowanie numeryczne, robotykę i administrowanie systemami. Informacje o ciekawych narzędziach i tematach znajdziesz w rozdziale 1. i swojej ulubionej przeglądarki.

W tym właśnie punkcie Python staje się prawdziwą zabawą, jednak tutaj także kończy się niniejsza książka, a zaczynają inne. Wskazówki dotyczące tego, gdzie warto się skierować po skończeniu lektury tej książki, znajdują się w „Przedmowie” w postaci listy polecanych tekstów. Mam nadzieję, że spotkamy się wkrótce w dziedzinie programowania aplikacji.

Powodzenia! I oczywiście: „Zawsze patrz na życie z humorem”!

Na bis: wydrukuj swój certyfikat!

Ostatnia rzecz: zamiast ćwiczeń do tej części książki umieściłem dodatkowy skrypt do samodzielnego przeanalizowania i uruchomienia. Nie mogę osobiście wręczać czytelnikom certyfikatów ukończenia lektury tej książki (byłyby jednak bezwartościowe, gdybym mógł), ale mogę zamieścić prosty skrypt w Pythonie, który to zrobi. Poniższy program, zawarty w pliku *certificate.py*, to skrypt przystosowany do wersji Pythona 2.x i 3.x, tworzący prosty certyfikat ukończenia kursu z książki w formacie tekstowym i HTML i wyświetlający go w domyślnej przeglądarce internetowej.

```
#!/usr/bin/python
"""

Plik certificate.py: skrypt dla Pythona 2.x i 3.x.

Generuje certyfikat ukończenia kursu, zapisuje w plikach w formacie tekstowym
i HTML oraz wyświetla w przeglądarce.

"""

from __future__ import print_function    # Kompatybilność z wersją 2.x
import time, datetime, sys, webbrowser, codecs
if sys.version_info[0] == 2:                # Kompatybilność z wersją 2.x
    input = raw_input
    import cgi
    htmlescape = cgi.escape
else:
    import html
    htmlescape = html.escape
maxline = 60                                # Separator wierszy
browser = True                               # Wyświetlenie w przeglądarce
saveto = 'Certificate.txt'                   # Nazwa pliku wyjściowego
template = """
%s
====> Oficjalny certyfikat <===
Data: %s
Niniejszy certyfikat potwierdza, iż
\t%s
przebrnął przez ogromny tom
\t%s
```

i ma teraz prawo do wszystkich swoich przywilejów, w tym do nauczania tworzenia stron internetowych, graficznych interfejsów użytkownika, modeli naukowych i różnorodnych aplikacji, z ewentualną pomocą kolejnych książek o aplikacjach, na przykład *Programming Python* (zamierzony, nieskromny wtręt).

--Mark Lutz, instruktor

(Uwaga: certyfikat jest nieważny, jeżeli zostały pominięte wszystkie rozdziałы.)

```
%s
"""

# Wprowadzenie danych, przygotowanie
for c in 'Gratulacje!'.upper():
    print(c, end=' ')
    sys.stdout.flush()                                # Niektóre powłoki mogą czekać
    na znak końca wiersza
    time.sleep(0.25)

print()
date = datetime.date.today()
name = input('Twoje imię i nazwisko: ').strip() or 'Nieznany czytelnik'
sept = '*' * maxline
book = 'Python. Wprowadzenie. Wydanie V'

# Utworzenie pliku tekstowego
file = codecs.open(saveto, 'w', 'utf8')
text = template % (sept, date, name, book, sept)
print(text, file=file)
file.close()

# Utworzenie pliku HTML
htmlto = saveto.replace('.txt', '.html')
file = codecs.open(htmlto, 'w', 'utf8')
tags = text.replace(sept, '<hr>')                 # Wstawienie kilku tagów
tags = tags.replace('==>', '<h1 align=center>')
tags = tags.replace('<==>', '</h1>')
tags = tags.split('\n')                               # Podzielenie na osobne
wiersze
tags = ['<center><p>' if line == ''
        else line for line in tags]
tags = ['<i>%s</i>' % htmlescape(line) if line[:1] == '\t'
```

```

        else line for line in tags]

tags = '\n'.join(tags)

link = '<i><a href="https://helion.pl">Wydawnictwo Helion</a></i>\n'

foot = '<table>\n<td>\n<td>%s</table></center>\n' % link

tags = '<html><body bgcolor=beige>' + tags + foot + '</body></html>'

print(tags, file=file)

file.close()

# Wyświetlenie wyniku

print('[File: %s]' % saveto, end='')

print('\n' * 2, open(saveto).read())

if browser:

    webbrowser.open(saveto, new=True)

    webbrowser.open(htmlto, new=False)

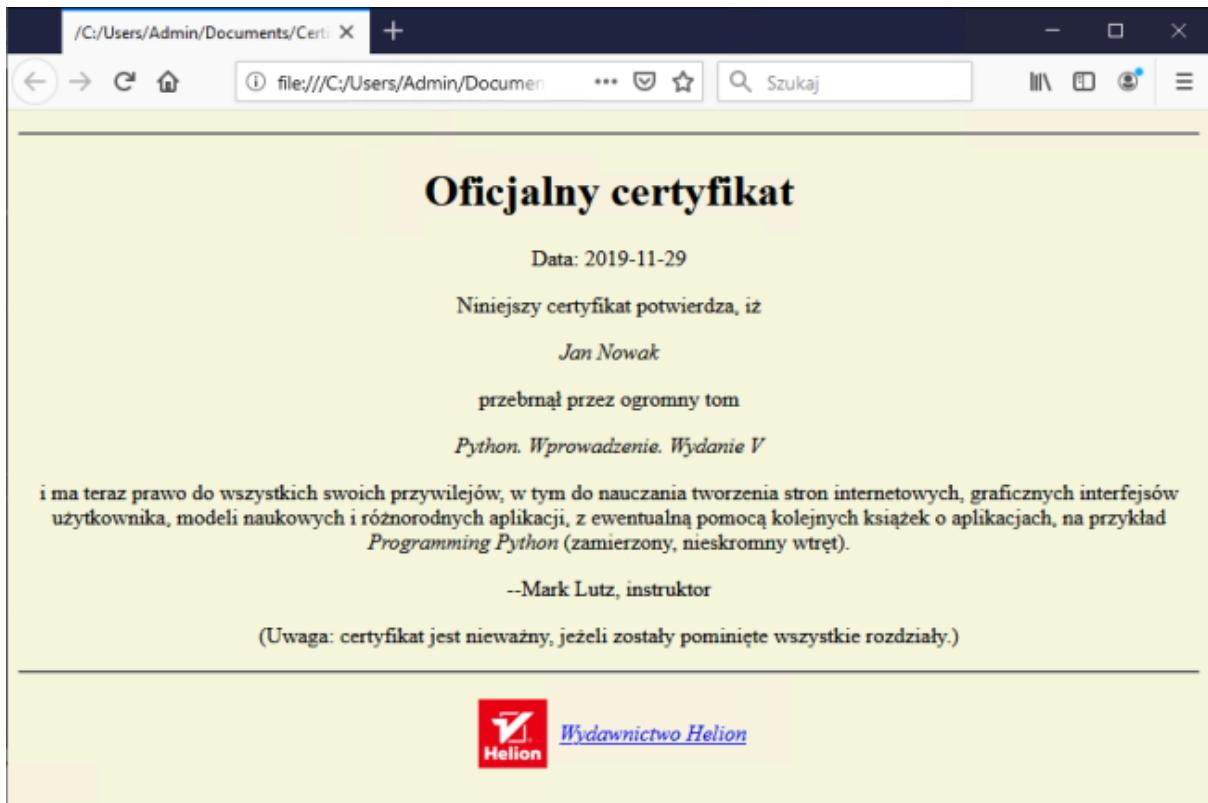
if sys.platform.startswith('win'):

    input('[Naciśnij Enter]')                      # W systemie Windows:  
pozostawienie otwartego okna

```

Uruchom samodzielnie powyższy skrypt i przeanalizuj kod, aby przypomnieć sobie niektóre zagadnienia opisane w tej książce. Skrypt możesz pobrać ze strony wskazanej w przedmowie. W tym kodzie nie ma żadnych deskryptorów, dekoratorów, metaklas ani klas nadzędnych, niemniej jednak jest to typowy kod Pythona.

Skrypt po uruchomieniu tworzy stronę pokazaną na rysunku 41.1. Oczywiście certyfikat może być o wiele bardziej okazały. Poszukaj w internecie informacji o tworzeniu dokumentów PDF i o innych narzędziach, np. o Sphinksie opisany w rozdziale 15. Skoro dotarłeś do końca książki, zasługujesz na dodatkowy żart lub dwa...



Rysunek 41.1. Strona WWW utworzona i otwarta za pomocą skryptu certificate.py

Dodatki

Dodatek A Instalacja i konfiguracja

W niniejszym dodatku znajdują się wskazówki dotyczące instalacji i konfiguracji Pythona, które mogą być pomocne nowicjuszom. Powstał dlatego, że nie wszyscy czytelnicy potrzebują tych informacji. Ponieważ zawarty tu materiał dotyczy pobocznych zagadnień, takich jak zmienne środowiskowe czy argumenty poleceń, większości czytelników wystarczy jego pobiczna lektura.

Instalowanie interpretera Pythona

Ponieważ do wykonywania skryptów Pythona potrzebny jest interpreter tego języka, pierwszym krokiem do używania Pythona jest zazwyczaj zainstalowanie go. O ile Python nie jest jeszcze dostępny na naszym komputerze, będziemy musieli pobrać, zainstalować i prawdopodobnie skonfigurować jakąś aktualną wersję. Wykonuje się to na każdym komputerze tylko raz, a jeśli będziemy wykonywać zamrożone pliki binarne (opisane w rozdziale 2.) bądź korzystać z samoinstalującego się systemu, być może nie będziemy musieli robić nic więcej.

Czy Python jest już zainstalowany?

Zanim zrobimy cokolwiek innego, należy sprawdzić, czy nie mamy już na komputerze zainstalowanej jakiejś aktualnej wersji Pythona. Osoby pracujące na systemach operacyjnych Linux, macOS czy niektórych systemach uniksowych mają najprawdopodobniej Pythona już zainstalowanego, choć może on być o jedno czy dwa wydania starszy od najbliższego. Oto, jak można to sprawdzić:

- W systemie Windows należy sprawdzić, czy w menu *Wszystkie programy* przycisku *Start* (po lewej stronie ekranu na dole) nie ma wpisu *Python*. W systemie Windows 8 należy odszukać kafelek *Python* lub kliknąć ikonę *Wszystkie aplikacje*. Ewentualnie można użyć narzędzia wyszukiwania i wpisać słowo *Python* (więcej informacji o systemie Windows 8 jest zawartych w ramce niżej).
- W systemie macOS należy otworzyć okno Terminala (*Aplikacje/Narzędzia/Terminal*) i w powłoce wpisać *python*. Standardowymi komponentami są: interpreter Pythona, środowisko IDLE i graficzne narzędzie tkinter.
- W systemach Linux i Unix należy wpisać *python* w powłoce systemowej (inaczej nazywanej oknem terminala) w celu sprawdzenia, co się stanie. Alternatywnie można spróbować wyszukać słowa „*python*” w typowych lokalizacjach — */usr/bin*, */usr/local/bin* i podobnych. Python jest standardową częścią systemu macOS i nie trzeba go instalować.

Jeśli znajdziemy Pythona, należy się upewnić, że jest to najnowsza wersja. Choć na potrzeby większej części niniejszej książki wystarczy dowolna aktualna wersja, to wydanie koncentruje się na Pythonie 3.3 i 2.7, dlatego by wykonać niektóre przykłady z książki, potrzebna będzie jedna z tych wersji.

A skoro już mowa o wersjach, osobom uczącym się Pythona od podstaw, które nie muszą pracować z istniejącym kodem w wersji 2.x, polecam zacząć od wersji 3.3 lub nowszej. W innym

przypadku należy korzystać z Pythona 2.7. Niektóre popularne systemy oparte na Pythonie nadal wykorzystują jeszcze starsze wydania (wersje 2.6 i 2.5 są ciągle bardzo rozpowszechnione), dlatego osoby pracujące z istniejącymi systemami muszą się upewnić, że korzystają z wersji odpowiadającej ich potrzebom. Poniżej opisane są miejsca, z których można pobrać różne wersje Pythona.

Skąd pobrać Pythona

Jeśli na naszym komputerze Python nie jest zainstalowany, będziemy musieli sami go zainstalować. Dobra wiadomość jest taka, że Python jest systemem *open source*, dostępnym za darmo w internecie i na większości platform bardzo łatwym do zainstalowania.

Najnowsze i najlepsze *standardowe wydanie Pythona* można zawsze pobrać ze strony <http://www.python.org>, oficjalnej witryny internetowej Pythona. Na tej stronie należy szukać odnośnika *Downloads*; później wystarczy wybrać platformę, na której będziemy pracować. Można tam znaleźć między innymi gotowy instalator przeznaczony dla systemu *Windows* (w celu zainstalowania należy go wykonać), pliki *Installer Disk Images* dla systemu *macOS* (instalowane zgodnie z regułami komputerów Mac) czy dystrybucje pełnych kodów źródłowych (zazwyczaj komplikowane w systemach *Linux*, *Unix* lub *macOS* w celu wygenerowania interpretera).

Choć Python jest obecnie w systemie *Linux* standardem, w internecie można także znaleźć pliki *RPM* przeznaczone dla tego systemu (należy je rozpakować za pomocą narzędzia *rpm*). Na stronie internetowej Pythona znajdują się również odnośniki do stron zewnętrznych, na których można znaleźć utrzymywane przez *Python.org* lub podmioty zewnętrzne wersje przeznaczone dla innych platform. Można na przykład znaleźć wersje instalacyjne Pythona dla systemu *Android* i *macOS* dla urządzeń przenośnych.

Pakiety Pythona można także znaleźć za pomocą wyszukiarki internetowej Google. Wśród innych platform znajdziemy gotowe pliki Pythona przeznaczone dla urządzeń takich, jak *iPod*, *Palm*, telefony komórkowe firmy *Nokia*, *PlayStation* i *PSP*, systemy *Solaris*, *AS/400* czy *Windows Mobile*.

Osoby troskliwe za środowiskiem Uniksa na komputerze z systemem Windows mogą także chcieć zainstalować bibliotekę *Cygwin* wraz z jej wersją Pythona (więcej informacji na ten temat można znaleźć na stronie <http://www.cygwin.com>). Cygwin to biblioteka i zbiór narzędzi na licencji *GPL*, udostępniający pełną funkcjonalność Uniksa na komputerach z systemem Windows. Zawiera także gotowego Pythona, który korzysta ze wszystkich udostępnionych narzędzi Uniksa.

Pythona można również znaleźć na płytach CD udostępnianych z dystrybucjami Linuksa jako część produktów czy systemów komputerowych, a także w niektórych książkach poświęconych temu językowi programowania. Zazwyczaj wersje z płyt są nieco w tyle za aktualnym wydaniem, choć nie aż tak bardzo.

Dodatkowo Pythona można znaleźć w niektórych pakietach programistycznych, zarówno darmowych, jak i komercyjnych. W chwili pisania tej książki dostępne były następujące *alternatywne dystrybucje*:

ActivePython firmy *ActiveState*

Pakiet zawierający standardową implementację Pythona, rozszerzenia zastosowań naukowych i programowania w systemie Windows (PyWin32) oraz zintegrowane środowisko programistyczne PythonWin.

Enthought Python Distribution

Pakiet zawierający interpreter Pythona oraz mnóstwo bibliotek i narzędzi przeznaczonych do zastosowań naukowych.

Portable Python

Odmiana Pythona wraz z dodatkowymi pakietami przygotowana do bezpośredniego uruchamiania na urządzeniach przenośnych.

Pythonxy

Dystrybucja Pythona przeznaczona do zastosowań naukowych, oparta na pakietach Qt i Spyder.

Conceptive Python SDK

Pakiet przeznaczony do tworzenia aplikacji biznesowych, stacjonarnych i bazodanowych.

PyIMSL Studio

Komercyjna dystrybucja przeznaczona do wykonywania analiz numerycznych.

Anaconda Python

Dystrybucja przeznaczona do analizowania i wizualizowania dużych zbiorów danych.

Powysza lista nieustannie się zmienia, dlatego szczegółowych informacji o powyższych i innych pakietach należy szukać w internecie. Niektóre dystrybucje są dostępne bezpłatnie, inne zarówno w wersjach płatnych, jak i darmowych. Wszystkie zawierają standardowego, bezpłatnego Pythona dostępnego na stronie <http://www.python.org>, a oprócz tego dodatkowe narzędzia ułatwiające instalację.

Wreszcie jeśli interesują nas alternatywne implementacje Pythona, powinniśmy wyszukać w internecie opisanych w rozdziale 2. implementacji *Jython* (Pythona przeznaczonego dla środowiska języka Java) oraz *IronPython* (dla świata .NET i języka C#). Instalacja tych dwóch systemów wykracza poza zakres niniejszej książki.

Instalacja Pythona

Po pobraniu Pythona będziemy muścieli go zainstalować. Poszczególne etapy instalacji są specyficzne dla określonej platformy, poniżej znajduje się jednak kilka wskazówek przeznaczonych dla najważniejszych środowisk, w jakich Python jest wykorzystywany (najszerzej opisana jest instalacja w systemie Windows, ponieważ jest to platforma, z której poczynając użytkownicy korzystają najczęściej):

Windows

W systemie Windows Python pobierany jest jako samoinstalujący plik programu MSI. Wystarczy tylko dwukrotnie kliknąć jego ikonę i odpowiadać Yes (Tak) lub Next (Dalej) na każde pytanie, by wykonać domyślną instalację. Domyślana instalacja zawiera zbiór dokumentacji Pythona oraz obsługę graficznego interfejsu użytkownika tkinter (w Pythonie 2.x Tkinter), baz danych modułu shelve, a także środowisko programistyczne IDLE. Python 3.3 i 2.7 są normalnie instalowane w katalogach *C:\Python33* oraz *C:\Python27*, choć można to zmienić w czasie instalacji.

Dla wygody po zainstalowaniu Python pojawi się w menu *Wszystkie programy* dostępnym pod przyciskiem *Start*. Menu Pythona zawiera pięć opcji umożliwiających szybki dostęp do często wykonywanych zadań — uruchomienia interfejsu użytkownika IDLE, odczytania dokumentacji modułu, uruchomienia sesji interaktywnej, wczytania dokumentacji biblioteki standardowej do przeglądarki oraz odinstalowania. Większość tych opcji obejmuje koncepcje omówione szczegółowo w niniejszej książce.

Po zainstalowaniu w systemie Windows domyślnie Python automatycznie rejestruje się jako program otwierający pliki Pythona po kliknięciu ich ikon (technika uruchamiania programów opisana w rozdziale 3.). Można również w systemie Windows zbudować

Pythona z kodu źródłowego, jednak nie jest to często stosowane, dlatego pominiemy tu szczegóły na ten temat (dostępne są na stronie python.org).

Dwie uwagi dotyczące użytkowników systemu Windows: po pierwsze należy zapoznać się z opisany w następującym dodatku nowym *programem uruchomieniowym* dostarczonym wraz z wersją 3.3. Zmienia on niektóre zasady instalacji, kojarzenia plików i używania wiersza poleceń, co ma swoje zalety, jeżeli w systemie zainstalowanych jest kilka wersji Pythona (np. 2.x i 3.x). Zgodnie z opisem w dodatku B program instalacyjny MSI można uruchomić z opcją powodującą dodanie katalogu Pythona do zmiennej środowiskowej PATH.

Po drugie użytkownicy systemu *Windows 8* powinni zapoznać się z ramką „Używanie Pythona w systemie Windows 8” w dalszej części rozdziału. Standardowy Python instaluje się i działa tak samo w systemie Windows 8, jak i w innych wersjach, z tą różnicą, że nie jest dostępny przycisk *Start*, a interfejs tabletu nie jest bezpośrednio obsługiwany.

Linux

W systemie Linux Python dostępny jest jako jeden lub większa liczba plików RPM, które rozpakowuje się w normalny sposób (po szczegóły należy sięgnąć do dokumentacji RPM). W zależności od wybranych plików RPM możemy otrzymać jeden z samym Pythonem i kolejne dodające obsługę graficznego interfejsu użytkownika `tkinter` oraz środowiska IDLE. Ponieważ Linux jest systemem podobnym do Unixa, poniższy akapit ma również zastosowanie do tego systemu.

Unix

W systemach uniksowych Python jest zazwyczaj komplikowany z pełnej dystrybucji kodu źródłowego w języku C. Zwykle wymaga to jedynie rozpakowania pliku i wykonania prostych poleceń `configure` oraz `make`. Python automatycznie konfiguruje swoją procedurę budowania zgodnie z systemem, na jakim jest komplikowany.Więcej informacji na temat tego procesu można znaleźć w pliku *README* pakietu. Ponieważ Python jest produktem open source, jego kod źródłowy może być wykorzystywany i dystrybuowany za darmo.

W przypadku innych platform szczegóły instalacji mogą różnić się między sobą, jednak zazwyczaj zgodne są z konwencjami dla danej platformy. Instalacja *Pippy*, wersji Pythona przeznaczonej dla systemu PalmOS, wymaga na przykład operacji synchronizacji typu HotSync. Python dla opartej na Linuksie serii PDA o nazwie Zaurus, produkowanych przez firmę Sharp, ma postać jednego lub większej liczby plików *.ipk*, które uruchamia się w celu zainstalowania (wersje te wciąż działają, choć znalezienie dzisiaj odpowiedniego urządzenia może być nie lada wyzwaniem).

Od niedawna Pythona można zainstalować i używać na urządzeniach z systemami *Android* i *macOS*, jednak wymaga to wykonania specyficznych dla tych systemów operacji i zastosowania odpowiednich technik, które nie zostały tu opisane. Szczegółowe procedury instalacyjne i najnowsze wersje można znaleźć na stronie Pythona i w internecie za pomocą wyszukiwarki.

Konfiguracja Pythona

Po zainstalowaniu Pythona możemy chcieć skonfigurować pewne ustawienia systemowe wpływające na sposób wykonywania kodu przez Pythona. (Jeśli dopiero zaczynamy swoją przygodę z tym językiem, możemy prawdopodobnie całkowicie pominąć ten podrozdział. W przypadku podstawowych programów nie ma najczęściej potrzeby wprowadzania zmian do ustawień systemowych).

Mówiąc ogólnie, część sposobu działania interpretera Pythona można skonfigurować za pomocą ustawień zmiennych środowiskowych oraz opcji wiersza poleceń. W niniejszym podrozdziale

przyjrzymy się krótko obu opcjom, jednak należy sprawdzić inne źródła dokumentacji w celu uzyskania szczegółowych informacji na przedstawione tutaj tematy.

Zmienne środowiskowe Pythona

Zmienne środowiskowe, znane również jako zmienne powłoki lub zmienne DOS, są ustawieniami systemowymi, które znajdują się poza Pythonem i tym samym mogą być wykorzystywane do dostosowania interpretera do naszych wymagań za każdym razem, gdy jest on uruchamiany na danym komputerze. Python rozpoznaje pewną liczbę ustawień zmiennych środowiskowych, jednak tylko niektóre z nich są używane na tyle często, by należało je tu omówić. W tabeli A.1 przedstawiono najważniejsze zmienne środowiskowe związane z Pythonem (informacje o innych są dostępne w materiałach dotyczących tego języka).

Tabela A.1. Istotne zmienne środowiskowe

Zmienna	Rola
PATH (lub path)	Systemowa ścieżka wyszukiwania powłoki (służąca do odnalezienia słowa python)
PYTHONPATH	Ścieżka wyszukiwania modułów Pythona (importowanie)
PYTHONSTARTUP	Ścieżka do interaktywnego pliku uruchomieniowego
TCL_LIBRARY, TK_LIBRARY	Zmienne rozszerzeń GUI (tkinter)
PY PYTHON, PY PYTHON3, PY PYTHON2	Domyślne ustawienia programu uruchamiającego Pythona (patrz dodatek B)

Powыższe zmienne środowiskowe są proste w użyciu, jednak warto umieścić tu kilka wskazówek.

PATH

Ustawienie PATH wymienia zbiór katalogów, które system operacyjny przeszukuje pod kątem programów uruchamianych bez podania pełnej ścieżki. Normalnie zmienna ta powinna zawierać katalog, w którym znajduje się nasz interpreter Pythona (program *python* w systemie Unix lub plik *python.exe* w systemie Windows).

Nie musimy ustawać tej zmiennej, jeśli zamierzamy pracować w katalogu, w którym Python się znajduje, lub wpisywać pełną ścieżkę do Pythona w wierszu poleceń. W systemie Windows na przykład zmienna PATH nie ma znaczenia, jeśli wykonamy polecenie `cd C:\Python33` przed wykonaniem jakiegokolwiek kodu (w celu zmiany na katalog, w którym znajduje się Python, choć zgodnie z rozdziałem 3. nie należy w tym katalogu zapisywać własnych kodów) lub zawsze zamiast samego *python* będziemy wpisywać `C:\Python33\python` (podając pełną ścieżkę).

Warto również zauważyć, że ustawienia zmiennej środowiskowej PATH służą przede wszystkim do uruchamiania programów z wiersza poleceń. Zazwyczaj nie mają znaczenia, kiedy programy uruchamia się za pomocą klikania ich ikon lub za pośrednictwem środowisk programistycznych. W pierwszym przypadku wykorzystywane są skojarzenia rozszerzeń nazw plików, a w drugim wbudowany mechanizm. Nie są przy tym wymagane żadne operacje konfiguracyjne. Szczegóły dotyczące automatycznego modyfikowania zmiennej PATH podczas instalacji są również opisane w dodatku B.

PYTHONPATH

Ustawienie PYTHONPATH pełni rolę podobną do PATH. Interpreter Pythona konsultuje się ze zmienną PYTHONPATH w celu zlokalizowania plików modułów, kiedy się je *importuje* w programie. Jeśli korzystamy z tej zmiennej, zostaje ona ustawiona na listę nazw katalogów w postaci specyficznej dla określonej platformy, rozdzielonych w systemie Unix za pomocą dwukropka, a w systemie Windows — średnika. Lista ta normalnie zawiera tylko nasze własne katalogi z kodem źródłowym. Jej zawartość łączona jest ze ścieżką wyszukiwania importowanych modułów sys.path, wraz z katalogiem skryptu, ustawieniami z plików ścieżek i katalogami biblioteki standardowej.

Nie musimy ustawiać tej zmiennej, o ile nie wykonujemy operacji *importowania pomiędzy różnymi katalogami*. Ponieważ Python zawsze automatycznie przeszukuje katalog główny pliku najwyższego poziomu programu, to ustawienie jest niezbędne tylko wtedy, gdy moduł potrzebuje zaimportować inny moduł znajdujący się w innym katalogu. Warto zobaczyć również omówienie plików ścieżek .pth w dalszej części dodatku, ponieważ są one alternatywą dla zmiennej PYTHONPATH. Więcej informacji na temat ścieżki wyszukiwania modułów znajduje się w rozdziale 22.

PYTHONSTARTUP

Jeśli zmienna PYTHONSTARTUP zostanie ustawiona na ścieżkę pliku z kodem Pythona, Python wykonuje ten kod pliku automatycznie za każdym razem, gdy uruchamiamy interpreter interaktywny, tak jakbyśmy wpisali wywołanie pliku w wierszu sesji interaktywnej. Jest to wykorzystywane rzadko, jednak przydaje się do upewnienia się, że zawsze ładujemy pewne narzędzia w pracy interaktywnej. Oszczędza nam to konieczności importowania.

Ustawienia tkinter

Jeśli chcemy skorzystać z zestawu narzędzi graficznego interfejsu użytkownika tkinter (w Pythonie 2.6 występującego pod nazwą Tkinter), być może konieczne będzie ustawienie dwóch zmiennych GUI z ostatniego wiersza tabeli A.1 na nazwy katalogów biblioteki źródeł systemów Tcl oraz Tk (podobnie jak robi się to w przypadku PYTHONPATH). Te ustawienia nie są jednak wymagane w systemie Windows (gdzie obsługa tkinter jest instalowana wraz z Pythonem) i zazwyczaj nie są konieczne w systemach macOS i Linux, chyba że biblioteki Tcl i Tk są albo niepoprawne, albo znajdują się w niestandardowych katalogach (więcej informacji na ten temat jest dostępnych na stronie python.org w sekcji *Download*).

PY PYTHON, PY PYTHON3, PY PYTHON2

Te zmienne służą do określania domyślnej wersji Pythona wykorzystywanej przez nowy (w chwili pisania tej książki) program uruchamiający w systemie Windows. Program ten jest dostarczany wraz z wersją 3.3, a dla starszych wersji jest dostępny osobno. Szczegółowe informacje na ten temat są zawarte w dodatku B, więc tutaj je pomijamy.

Warto zauważyc, że ponieważ ustawienia zmiennych środowiskowych są zewnętrzne dla samego Pythona, moment skonfigurowania ich nie ma zazwyczaj znaczenia. Można to zrobić przed instalacją Pythona lub już po niej; wystarczy pamiętać, że ustawienia powinny mieć odpowiednie wartości przed samym *wykonaniem* Pythona.

Obsługa graficznego interfejsu użytkownika tkinter oraz IDLE w systemach Linux i macOS

Interfejs IDLE opisany w rozdziale 3. jest programem napisanym w Pythonie i opartym na graficznym interfejsie użytkownika tkinter. Moduł tkinter (w Pythonie 2.x znany jako Tkinter) to zestaw narzędzi GUI, który jest kompletnym, standardowym komponentem Pythona w systemie Windows i nieodłączną częścią systemów macOS i Linux.

W niektórych systemach Linux biblioteka GUI będąca jego podstawą może nie być standardowym, zainstalowanym komponentem. By dodać obsługę graficznego interfejsu użytkownika do Pythona zainstalowanego na Linuksie, można spróbować wpisać w wierszu poleceń yum install tkinter, by automatycznie zainstalować biblioteki będące podstawą tkinter. Powinno to działać we wszystkich dystrybucjach Linuksa (i niektórych

innych systemach), w których dostępny jest program instalacyjny yum. W innych systemach należy użyć narzędzi instancyjnych opisanych w dokumentacji. Jak wspomniano w rozdziale 3., w systemie macOS środowisko IDLE jest umieszczane w folderze *Aplikacje* i podfolderze *MacPython* (lub *Python N.M*) wraz z programem *PythonLauncher* wykorzystywanym do uruchamiania programów za pomocą narzędzia Finder. Jeżeli pojawią się problemy ze środowiskiem IDLE, należy zapoznać się z informacjami dostępnymi na stronie python.org w sekcji *Download*. W niektórych systemach macOS może być konieczne zainstalowanie aktualizacji (patrz rozdział 3.).

Jak ustawić opcje konfiguracyjne?

Metoda ustawiania zmiennych środowiskowych związanych z Pythonem, a także to, na co je należy ustawić, zależy od typu komputera, na jakim pracujemy. I znów należy pamiętać, że nie musimy koniecznie ustawiać zmiennych środowiskowych od razu — w szczególności kiedy pracujemy w IDLE (aplikacji opisanej w rozdziale 3.), ich konfiguracja nie jest wymagana z góry.

Załóżmy jednak, dla celów ilustracyjnych, że w katalogach *utilities* oraz *package1* gdzieś na komputerze mamy uniwersalnie przydatne pliki modułów, które chcemy móc importować do plików umieszczonych w innych katalogach. Żeby na przykład załadować plik *spam.py* z katalogu *utilities*, chcemy móc napisać po prostu:

```
import spam
```

w innym pliku znajdującym się w dowolnym miejscu komputera. By to zadziałało, musimy skonfigurować ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona katalog zawierający plik *spam.py*. Poniżej znajduje się kilka wskazówek dotyczących tego procesu na przykładzie zmiennej *PYTHONPATH*. Można je też zastosować do zmiennej *PATH*, jeżeli zajdzie taka potrzeba (choć w wersji 3.3 zmienność *PATH* jest ustawiana automatycznie — patrz dodatek B).

Zmienne powłoki systemu Unix i Linux

W systemach Unix sposób ustawienia zmiennych środowiskowych uzależniony jest od wykorzystywanej powłoki. W przypadku powłoki *csh* możemy do pliku *.cshrc* lub *.login* dodać poniższy wiersz, by ustawić ścieżkę wyszukiwania modułów Pythona.

```
setenv PYTHONPATH /usr/home/pycode/utilities:/usr/lib/pycode/package1
```

Polecenie to każe Pythonowi szukać importowanych modułów w dwóch katalogach zdefiniowanych przez użytkownika. Alternatywnie, jeśli korzystamy z powłoki *ksh*, ustawienie to może zamiast tego pojawić się w pliku *.kshrc* i wyglądać tak, jak poniższy kod.

```
export PYTHONPATH="/usr/home/pycode/utilities:/usr/lib/pycode/package1"
```

Inne powłoki mogą wykorzystywać odmienną (choć analogiczną) składnię.

Zmienne DOS (system Windows)

Jeśli korzystamy z systemu MS-DOS lub starszych odmian systemu Windows, aby móc będącym muśnięci dodać polecenie konfigurujące zmienną środowiskową do pliku *C:\autoexec.bat* i uruchomić ponownie komputer, tak by zmiany zaczęły działać. Polecenie konfiguracyjne w takich komputerach wykorzystuje składnię unikalną dla systemu DOS.

```
set PYTHONPATH=c:\pycode\utilities;d:\pycode\package1
```

Takie polecenie można również wpisać w oknie konsoli DOS, jednak ustawienie takie będzie wtedy aktywne jedynie dla tego jednego okna konsoli. Modyfikacja pliku *.bat* sprawia, że

zmiana będzie stała i globalna dla wszystkich programów. Powyższa technika została jednak w ostatnich latach zastąpiona inną, opisaną niżej.

Graficzny interfejs użytkownika zmiennych środowiskowych Windows

W nowszych wersjach systemu Windows (w tym Windows 8, 8.1 i 10), możemy zamiast tego ustawić PYTHONPATH oraz inne zmienne za pomocą graficznego interfejsu użytkownika systemowych zmiennych komputera bez konieczności edytowania plików, wpisywania poleceń i ponownego uruchamiania komputera. Aby zmienić lub utworzyć nową zmienną środowiskową, należy otworzyć Panel sterowania (dostępny po kliknięciu przycisku *Start*), kliknąć ikonę *System*, odnośnik *Zaawansowane ustawienia systemu* i przycisk *Zmienne środowiskowe* w dolnej części okna. PYTHONPATH jest zazwyczaj nową zmienną użytkownika. Należy użyć tych samych nazw zmiennych i składni wartości co zaprezentowane wcześniej w poleceniu DOS set.

Nie trzeba uruchamiać ponownie komputera, jednak należy pamiętać o ponowym uruchomieniu Pythona, jeśli jest on otwarty, tak by pobrał on wprowadzone zmiany (ścieżka konfigurowana jest jedynie w czasie uruchamiania Pythona). Jeśli pracujemy w oknie *Wiersza poleceń* systemu Windows, będziemy najprawdopodobniej musieli ponownie uruchomić również ten program.

Rejestr systemu Windows

Zaawansowani użytkownicy tego systemu operacyjnego mogą również skonfigurować ścieżkę wyszukiwania modułów za pomocą *Edytora rejestru*. Należy wybrać z menu *Start* opcję *Uruchom...* i wpisać *regedit*. Zakładając, że na naszym komputerze znajduje się typowe narzędzie do obsługi rejestru, możemy teraz przejść do wpisów dotyczących Pythona i wprowadzić do nich odpowiednie zmiany. Jest to jednak procedura delikatna i podatna na błędy, dlatego osobom niezaznajomionym z rejestrem systemu Windows odradzam jej stosowanie. Właściwie przypomina to trochę operację na mózgu komputera, dlatego należy bardzo uważać.

Pliki ścieżek

Wreszcie jeśli zdecydujemy się rozszerzyć ścieżkę wyszukiwania modułów za pomocą pliku *.pth* zamiast zmiennej PYTHONPATH, możemy zamiast tego utworzyć plik tekstowy przypominający poniższy z systemu Windows (plik *C:\Python33\mypath.pth*).

```
c:\pycode\utilities  
d:\pycode\package1
```

Jego zawartość będzie różna dla różnych platform, a katalog zawierający może się zmieniać zarówno dla różnych systemów operacyjnych, jak i wersji Pythona. Python lokalizuje ten plik automatycznie w momencie uruchomienia.

Nazwy katalogów w plikach ścieżki mogą być bezwzględne lub określane względem katalogu zawierającego plik ścieżki. Można stosować większą liczbę plików *.pth* (dodane zostaną wszystkie ich katalogi), a same pliki *.pth* mogą się pojawiać w różnych automatycznie sprawdzanych katalogach specyficznych dla platformy i wersji Pythona. Generalnie dla wersji Pythona oznaczonej jako *N.M* pliki ścieżek szukane są zazwyczaj w katalogach *C:\PythonNM* oraz *C:\PythonNM\Lib\site-packages* w systemie Windows i w */usr/local/lib/pythonN.M/site-packages* oraz */usr/local/lib/site-python* w systemach Unix oraz Linux. Więcej informacji na temat wykorzystywania plików ścieżek w celu konfiguracji ścieżki wyszukiwania importowanych modułów sys.path można znaleźć w rozdziale 22.

Ponieważ ustawienia środowiskowe są często opcjonalne, a także dlatego, że książka ta nie jest poświęcona powłokom systemowym, po pozostałe szczegóły odsyłam do innych źródeł. Należy sięgnąć do dokumentacji powłoki systemowej, a jeśli mamy kłopot z odpowiednimi ustawieniami, warto poprosić o pomoc administratora systemu czy innego lokalnego eksperta.

Opcje wiersza poleceń Pythona

Kiedy uruchamiamy Pythona z systemowego wiersza poleceń, możemy przekazać różne opcje umożliwiające kontrolowanie tego, w jaki sposób działa Python. W przeciwieństwie do ustawionych dla całego systemu zmiennych środowiskowych opcje wiersza poleceń mogą być inne za każdym razem, gdy wykonujemy skrypt. Pełna forma wywołania Pythona z wiersza poleceń w wersji 3.3 wygląda następująco (w 2.7 jest mniej więcej tak samo, różni się jedynie kilka opcji):

```
python [-bBdEhi0qsSuvVWxX] [-c polecenie | -m nazwa-modułu | skrypt | - ]  
[argumenty]
```

W pozostałą części tego podrozdziału krótko opisane są najczęściej stosowane argumenty. Więcej informacji na temat dostępnych opcji wiersza poleceń można znaleźć w dokumentacji Pythona lub innych tekstach. A jeszcze lepiej będzie zapytać o to samego Pythona — wykonując polecenie takie jak poniższe:

```
C:\code> python -h
```

w celu uzyskania wyświetlanego pomocy Pythona, dokumentującej dostępne opcje wiersza poleceń. Jeżeli polecenia są skomplikowane, warto skorzystać ze standardowych modułów. Oryginalny moduł `getopt`, nowszy `argparse` i wycofany już (wraz z pojawiением się wersji 3.2) `optparse` umożliwiają wpisywanie bardziej zaawansowanych polecen. Ponadto można skorzystać z dokumentacji i dodatkowych materiałów na temat modułów `pdb` i `profile`.

Uruchamianie skryptów z argumentami

Większość wierszy poleceń wykorzystuje jedynie części `skrypt` oraz `argumenty` z tego formatu w celu wykonania pliku źródłowego programu z argumentami, które mają być wykorzystane przez sam program. By to zilustrować, rozważmy poniższy skrypt. Jest to plik tekstowy o nazwie `showargs.py` umieszczony w katalogu `C:\code` lub dowolnym innym. Skrypt ten wyświetla argumenty wiersza poleceń dostępne w liście `sys.argv` (aby dowiedzieć się, jak tworzy się i uruchamia skrypty w Pythonie, należy zapoznać się z rozdziałami 2. i 3.; tutaj interesują nas tylko argumenty wiersza poleceń).

```
# Plik showargs.py  
  
import sys  
  
print(sys.argv)
```

W poniższym wierszu poleceń `python` oraz `showargs.py` mogą również być pełnymi ścieżkami do katalogów. Tutaj przyjęte zostało założenie, że ścieżka do programu została zapisana w zmiennej `PATH`, a plik znajduje się w bieżącym katalogu. Trzy argumenty (`a` `b` `-c`) przeznaczone dla skryptu pokazują się na liście `sys.argv` i skrypt ten je wyświetla. Pierwszym elementem `sys.argv` jest zawsze nazwa pliku skryptu, nawet jeśli jest ona znana.

```
c:\code> python showargs.py a b -c          # Najpopularniejsze rozwiązanie:  
wykonanie pliku skryptu  
  
['showargs.py', 'a', 'b', '-c']
```

Jak wspomniałem w tej książce, listy są wyświetlane w nawiasach kwadratowych, a ciągi znaków są ujęte w apostrofy.

Uruchamianie kodu podanego w argumentach i pobranego ze standardowego wejścia

Inne opcje formatu kodu pozwalają nam określić kod Pythona do wykonania w samym wierszu poleceń (`-c`), przyjmować kod do wykonania ze standardowego strumienia wejścia (`a` — oznacza wczytanie z potoku lub przekierowanego pliku strumienia wejścia) i tak dalej:

```

c:\code> python -c "print(2 ** 100)"      # Wczytanie kodu z argumentu
polecenia
1267650600228229401496703205376
c:\code> python -c "import showargs"      # Zimportowanie pliku w celu
wykonania jego kodu
['-c']
c:\code> python - < showargs.py a b -c    # Wczytanie kodu ze standardowego
wejścia
['-', 'a', 'b', '-c']
c:\code> python - a b -c < showargs.py    # Ten sam efekt co poprzedni wiersz
['-', 'a', 'b', '-c']

```

Uruchamianie modułów umieszczonych w ścieżce wyszukiwania

Opcja `-m` lokalizuje moduł w ścieżce wyszukiwania modułów Pythona (`sys.path`) i wykonuje go jako skrypt najwyższego poziomu (jako moduł `__main__`). Oznacza to, że szuka skryptu w taki sam sposób, jak instrukcja `import`, tj. wykorzystuje listę katalogów zapisanych w zmiennej `sys.path`. Lista ta zawiera bieżący katalog, zawartość zmiennej środowiskowej `PYTHONPATH` oraz ścieżki do standardowych bibliotek. Opuszczamy tutaj rozszerzenie `.py`, ponieważ nazwa pliku jest modułem.

```

c:\code> python -m main a b -c            # Zlokalizowanie i
wykonanie modułu jako skryptu
['c:\\Python30\\main.py', 'a', 'b', '-c']

```

Opcja `-m` obsługuje również wykonywanie modułów w pakietach za pomocą składni importowania względnego, w tym modułów znajdujących się w archiwach `.zip`. Jest często wykorzystywana do uruchamiania modułów debugera `pdb` i narzędzia profilującego `profile` z wiersza poleceń w celu wywołania skryptu zamiast wykonywania interaktywnego.

```

C:\code> python                         # Interaktywna sesja
debugera

>>> import pdb
>>> pdb.run('import showargs')
...reszta pominięta: patrz dokumentacja do pdb
c:\code> python -m pdb showargs.py a b -c      # Debugowanie skryptu
(c=kontynuacja)

> c:\code\showargs.py(2)<module>()
-> import sys
(Pdb) c
['showargs.py', 'a', 'b', '-c']
... reszta pominięta: naciśnij q, aby zakończyć

```

Poniższe polecenie uruchamia program profilujący, który mierzy czas wykonywania kodu. Wyniki mogą się różnić w zależności od wersji Pythona, systemu operacyjnego i szybkości komputera:

```
c:\code> python -m profile showargs.py a b -c      # Profilowanie skryptu
```

```

['showargs.py', 'a', 'b', '-c']

    9 function calls in 0.016 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall   filename:lineno(function)

    2    0.000    0.000      0.000    0.000 :0(charmap_encode)
    1    0.000    0.000      0.000    0.000 :0(exec)

... reszta pominięta: patrz dokumentacja do profile

```

Za pomocą opcji `-m` można w dowolnym katalogu uruchomić opisane w rozdziale 3. środowisko graficzne IDLE zawarte w standardowej bibliotece. Opcja ta służy również do uruchamiania za pomocą wiersza poleceń narzędzi `pydoc` i `timeit` opisanych szczegółowo w rozdziałach 15. i 21.:

```

c:\code> python -m idlelib.idle -n                      # Uruchomienie IDLE w
pakietie, bez podprocesów

c:\code> python -m pydoc -b                          # Uruchomienie narzędzi
pydoc i timeit tools

c:\code> python -m timeit -n 1000 -r 3 -s "L = [1,2,3,4,5]" "M = [x + 1 for x
in L]"

```

Tryby zoptymalizowany i niebuforowany

Natychmiast po słowie `python` i przed wyznaczeniem kodu, który ma być wykonany, Python przyjmuje dodatkowe argumenty kontrolujące jego własne działanie. Argumenty te są przez niego pobierane i nie są przeznaczone dla wykonywanego skryptu. Przykładowo `-O` uruchamia Pythona w trybie zoptymalizowanym, `-u` wymusza brak bufora dla standardowych strumieni. W efekcie wyświetlenie tekstu nie jest opóźniane z powodu umieszczenia go w buforze:

```

C:\code> python -O showargs.py a b -c          # Tryb zoptymalizowany:
utworzenie/uruchomienie kodu

# bajtowego ".pyo"

C:\code> python -u showargs.py a b -c          # Standardowe strumienie bez
bufora

```

Tryb interaktywny po wykonaniu skryptu

Opcja `-i` powoduje przejście do trybu interaktywnego po wykonaniu skryptu. Jest ona szczególnie przydatna jako narzędzie diagnostyczne, ponieważ umożliwia wyświetlanie wartości zmiennych i uzyskiwanie szczegółowych informacji po pomyślnym wykonaniu kodu:

```

C:\code> python -i showargs.py a b -c          # Przejście do trybu
interaktywnego po zakończeniu skryptu

['showargs.py', 'a', 'b', '-c']

>>> sys                                         # Ostateczna wartość zmiennej sys:
zimportowany moduł

<module 'sys' (built-in)>

>>> ^Z

```

W ten sposób można wyświetlać wartości zmiennych i sprawdzać stan programu po przerwaniu jego działania z powodu zgłoszenia wyjątku, nawet gdy nie zostanie użyty tryb diagnostyczny. Można też uruchomić w tym momencie debugera (type jest poleceniem wyświetlającym zawartość pliku w systemie Windows; w innych systemach należy użyć polecenia try lub cat):

```
C:\code> type divbad.py
X = 0
print(1 / X)
C:\code> python divbad.py                                # Uruchomienie błędnego
skryptu
...tekst błędu pominięty
ZeroDivisionError: division by zero
C:\code> python -i divbad.py                            # Wyświetlenie wartości
zmiennych po zgłoszeniu błędu
...tekst błędu pominięty
ZeroDivisionError: division by zero
>>> X
0
>>> import pdb                                         # Otwarcie pełnej sesji
diagnostycznej
>>> pdb.pm()
> C:\code\divbad.py(2)<module>()
-> print(1 / X)
(Pdb) quit
```

Argumenty wiersza poleceń w Pythonie w.x

Python 2.7 obsługuje dodatkowe opcje oprócz wyżej opisanych włączające zgodność z wersją 3.x (-3 ostrzega o niekompatybilności, -Q steruje modelami operatora dzielenia) oraz wykrywanie niespójnego użycia indentacji kodu, co w Pythonie 3.x jest zawsze wykrywane i zgłoszane (-t, patrz rozdział 12.). I jak poprzednio — zawsze można zapytać samego Pythona o więcej informacji na żądany temat, wpisując następujące polecenie:

```
C:\code> c:\python27\python -h
```

Uruchamianie Pythona 3.3 za pomocą wiersza poleceń Windows

W poprzednim podrozdziale zostały opisane argumenty, które można stosować w samym interpreterze Pythona, którym zazwyczaj jest program o nazwie *python.exe* w systemie Windows (rozszerzenie *.exe* zazwyczaj się pomija) i *python* w Linuksie. Jak się dowiemy w następnym dodatku, program uruchamiający Pythona w systemie Windows oferuje dodatkowe możliwości dla użytkowników standardowego pakietu uruchamiającego, jak i dostarczanego razem z wersją 3.3 lub nowszą. Dostępne są nowe pliki uruchamiane w wierszu poleceń, umożliwiające podanie w argumentach numeru wersji Pythona i nazwy uruchamianego skryptu

(tutaj skrypt *what.py*, opisany w następnym dodatku, wyświetla po prostu numer wersji Pythona):

```
C:\code> py what.py                                # Program uruchomieniowy w
systemie Windows

3.3.0

C:\code> py -2 what.py                            # Argument określający numer
wersji

2.7.3

C:\code> py -3.3 -i what.py -a -b -c          # Argumenty dla wszystkich
trzech komponentów: py, python i skryptu

3.3.0

>>> ^Z
```

Jak pokazuje ostatni przykład, w wierszu poleceń można umieścić argumenty dla programu uruchomieniowego (-3.3), samego Pythona (-i) oraz skryptu (-a, -b i -c). Ponadto program uruchomieniowy uwzględnia numer wersji podany na początku skryptu w wierszu `#!`. Ponieważ temu tematowi jest poświęcony następny dodatek, zachęcam do kontynuowania lektury.

Uzyskanie pomocy

Dzisiejszy zbiór dokumentacji Pythona zawiera wartościowe wskazówki dotyczące używania tego języka na różnych platformach. Dokumentacja biblioteki standardowej dostępna jest w systemie Windows pod menu *Start* od razu po zainstalowaniu Pythona (opcja *Python Manuals*), a także w internecie pod adresem <http://www.python.org>. Więcej wskazówek dotyczących wykorzystywania Pythona na różnych platformach, a także aktualne informacje dotyczące środowiska i wierszy poleceń na poszczególnych platformach można znaleźć w dziale *Using Python* dokumentacji.

Jak zawsze pomocy można także szukać w internecie, zwłaszcza w tej dziedzinie, która często zmienia się bardzo szybko — szybciej, niż można uaktualnić książki. Biorąc pod uwagę szerokie rozpowszechnienie Pythona, jest bardzo prawdopodobne, że odpowiedzi na wszystkie pytania dotyczące korzystania z tego języka znajdziemy za pomocą wyszukiwarki.

Dodatek B Uruchamianie Pythona 3.x w systemie Windows

W niniejszym dodatku opisany jest nowy program uruchamiający Pythona w systemie Windows, dostarczany razem z wersją 3.3, a dla starszych wersji dostępny do pobrania na stronie WWW. Program wprowadza dodatkowy poziom obsługi umożliwiający wybranie i uruchomienie interpretera Pythona. Choć ma kilka pułapek, zapewnia niezbędną spójność środowiska w przypadku, gdy w systemie zainstalowanych jest kilka wersji języka.

Niniejszy dodatek napisałem dla programistów używających Pythona w systemie Windows. Choć program uruchomieniowy jest związany z tym właśnie systemem, jest przeznaczony zarówno dla początkujących użytkowników (z których większość korzysta z tego systemu operacyjnego), jak również programistów tworzących uniwersalny kod dla systemów Windows i Unix. Jak się przekonamy, nowy program uruchomieniowy radykalnie zmienia zasady korzystania z Pythona w systemie Windows, które dotyczą wszystkich obecnych i przyszłych użytkowników.

Dziedzictwo systemu Unix

Aby w pełni zrozumieć protokół uruchamiania kodu, musimy przejść krótką lekcję historii. Dawno temu programiści opracowali dla systemu Unix (jak również Linux i macOS) protokół wybierający program do uruchamiania skryptu. Zgodnie z nim pierwszy wiersz zawierający sekwencję dwóch znaków `#!`, tzw. *shebang* (z ang. *kram* — przedziwne określenie, obiecuję go tutaj nie używać), jest traktowany w specjalny sposób.

W rozdziale 3. ten temat był krótko opisany, ale teraz zbadamy go dokładniej. W systemie Unix wspomniany wyżej wiersz wskazuje program, który ma wykonać pozostałą część kodu. Nazwę programu umieszcza się po znakach `#!`. Może nią być pełna ścieżka do pliku lub polecenie env wyszukujące w katalogach zawartych w zmiennej środowiskowej PATH docelowy plik wykonywalny:

```
#!/usr/local/bin/python
...kod skryptu           # Uruchomienie wskazanego programu
#!/usr/bin/env python
...kod skryptu           # Uruchomienie programu Python znalezioneego w
zmiennej PATH
```

Po oznaczeniu takiego skryptu jako wykonywalnego (np. za pomocą polecenia `chmod +x script.py`) można go uruchamiać, wpisując po prostu jego nazwę w wierszu polecen. Wiersz rozpoczynający się od `#!` kieruje powłokę Uniksa do programu, który wykona pozostałą część skryptu. W zależności od struktury systemu polecenie `python` w takim wierszu może być właściwym programem wykonywalnym lub łączem symbolicznym do określonej wersji programu zapisanego na dysku. Można również podać dokładną nazwę programu, np. `python3`.

Tak czy inaczej, zmieniając wiersz `#!`, łącze symboliczne i zmienną PATH, można w Uniksie przekazywać skrypt do odpowiedniej wersji Pythona zainstalowanej w systemie.

Oczywiście żaden z powyższych sposobów nie dotyczy systemu Windows, ponieważ wiersz `#!` nie ma w nim specjalnego znaczenia — interpreter Pythona sam z siebie pomija takie wiersze (znak `#` oznacza komentarz). Niemniej jednak możliwość wybierania pliku wykonywalnego Pythona indywidualnie dla każdego skryptu jest bardzo ważna w przypadku, gdy w tym samym systemie istnieją wersje 2.x i 3.x. Zważywszy, że w celu zapewnienia kompatybilności z systemem Unix wielu programistów wpisuje wiersz `#!`, cała związana z nim idea wydała się godna naśladowania.

Dziedzictwo systemu Windows

Po drugiej stronie systemowego muru model instalacji oprogramowania był zupełnie inny. W przeszłości (a dokładniej zanim pojawiła się wersja Pythona 3.3) program instalacyjny zmieniał globalny rejestr systemu Windows. Po kliknięciu nazwy skryptu lub wpisaniu jej w wierszu poleceń zawsze była uruchamiana najnowsza zainstalowana wersja Pythona.

Niektórzy użytkownicy systemu Windows wiedzą, że skojarzenia typów plików konfiguruje się w Panelu sterowania, w oknie *Programy domyślne*. Aby plik był wykonywalny, nie trzeba nadawać mu specjalnych uprawnień, jak w systemie Unix. W systemie Windows nie ma nawet takiej możliwości. Skojarzenie typu pliku z poleceniem wystarczy, aby można go było uruchamiać jak program.

Aby w takim modelu instalacji uruchomić skrypt z inną wersją Pythona niż najnowsza zainstalowana, trzeba w wierszu poleceń wpisywać pełną ścieżkę do żądanego programu lub ręcznie zmieniać skojarzenia typów plików. Można również wskazać domyślne polecenie `python` z żądaną wersji, odpowiednio zmieniając zmienną PATH. Jednak program instalacyjny nie ustawia jej, a ponadto takie rozwiązanie nie dotyczyło skryptów uruchamianych przez dwukrotne kliknięcie ikony.

Jest to naturalna zasada w systemie Windows (na przykład po kliknięciu pliku `.docx` otwierana jest najnowsza zainstalowana wersja programu Word) obowiązująca, od kiedy pojawił się Python dla tego systemu operacyjnego. Nie sprawdza się jednak idealnie, jeżeli w systemie zapisane są skrypty wymagające użycia różnych wersji Pythona, co ma miejsce coraz częściej, a nawet jest rzeczą normalną w czasach podwójnej instalacji wersji 2.x/3.x. Zanim pojawiła się wersja 3.3, uruchamianie różnych wersji Pythona w systemie Windows było uciążliwą operacją, szczególnie zniechęcającą dla początkujących użytkowników.

Wprowadzenie nowego programu uruchomieniowego w systemie Windows

Nowy program uruchomieniowy, dostarczany i instalowany automatycznie od wersji 3.3, jak również dostępny w postaci osobnego pakietu dla starszych wersji, rozwiązuje problemy wynikające z modelu instalacyjnego poprzez wprowadzenie dwóch plików wykonywalnych:

- `py.exe` dla programów konsolowych,
- `pyw.exe` dla pozostałych programów (zazwyczaj wykorzystujących interfejs graficzny).

Te dwa programy są kojarzone odpowiednio z plikami `.py` i `.pyw`. Ponadto programy te, podobnie jak oryginalny główny plik `python.exe` (którego nie deprecjonują), lecz w dużej mierze

zastępują), pozwalają bezpośrednio uruchamiać pliki z kodami bajtowymi. Oprócz tego mają następujące atuty:

- automatycznie otwierają pliki z kodami źródłowymi i bajtowymi po kliknięciu ikony lub wpisaniu nazwy w wierszu poleceń (dzięki skojarzeniu typów plików);
- są instalowane w ścieżce poszukiwań, nie wymagają modyfikowania zmiennej PATH ani wpisywania pełnej ścieżki w wierszu poleceń;
- umożliwiają wskazywanie wersji Pythona za pomocą parametru polecenia uruchamiającego skrypt lub interaktywną sesję;
- analizują wiersze `#!` charakterystyczne dla systemu Unix w celu określenia wersji Pythona, która ma być użyta do wykonania kodu.

W efekcie, jeżeli w systemie zainstalowanych jest kilka wersji Pythona, to dzięki nowemu programowi uruchomieniowemu nie jest się skazanym na używanie najnowszej wersji ani zmuszanym do jawnego wpisywania pełnych ścieżek w wierszu poleceń. Zamiast tego można jawnie wskazywać wersję indywidualnie dla każdego pliku lub polecenia, zarówno w pełnej, jak i częściowej formie. Poniżej znajduje się opis, jak to się robi:

1. Aby wybrać wersję dla jednego pliku, należy na jego początku wpisać jeden z poniższych komentarzy, podobnie jak w systemie Unix:

```
#!/python2  
#!/usr/bin/python2.7  
#!/usr/bin/env python3
```

2. Aby wybrać wersję w *bieżącym poleceniu*, należy wpisać jeden z następujących parametrów:

- `py -2 m.py`
- `py -2.7 m.py`
- `py -3 m.py`

Pierwszą z tych technik można traktować jako swego rodzaju dyrektywę deklarującą wersję Pythona przeznaczoną dla danego skryptu. Dyrektywa ta jest stosowana przez program uruchomieniowy przy każdym uruchomieniu skryptu, czy to za pomocą wiersza poleceń, czy przez kliknięcie ikony (poniżej przedstawione są różne warianty przykładowego pliku *script.py*):

```
#!/python3  
...  
... skrypt 3.x      # Uruchomienie z najnowszą zainstalowaną wersją 3.x  
...  
#!python2  
...  
... skrypt 2.x      # Uruchomienie z najnowszą zainstalowaną wersją 2.x  
...  
#!python2.6  
...  
... skrypt 2.6      # Uruchomienie tylko z wersją 2.6  
...
```

W systemie Windows polecenia wpisuje się w wierszu poleceń, który w tym dodatku jest oznaczony ciągiem zachęty `C:\code>`. Pierwsze poniższe polecenie ma taki sam efekt jak drugie i jak kliknięcie ikony, ponieważ wykorzystywane są tutaj skojarzenia typów plików:

```
C:\code> script.py      # Uruchomienie z wersją wskazaną w wierszu #!, jeżeli taki jest, albo z domyślną wersją
```

```
C:\code> py script.py    # To samo, tylko program py.exe jest uruchamiany jawnie
```

Alternatywnym rozwiązaniem jest zastosowanie *drugiej* opisanej techniki polegającej na wskazaniu wersji w argumencie polecenia:

```
C:\code> py -3 script.py    # Uruchomienie z najnowszą zainstalowaną wersją 3.x
```

```
C:\code> py -2 script.py    # Uruchomienie z najnowszą zainstalowaną wersją 2.x
```

```
C:\code> py -2.6 script.py   # Uruchomienie tylko z wersją 2.6
```

Poniższe polecenia uruchamiają skrypt, jak również otwierają interaktywną sesję interpretera (jeżeli nie zostanie podana nazwa skryptu):

```
C:\code> py -3          # Uruchomienie interaktywnej sesji z najnowszą zainstalowaną wersją 3.x
```

```
C:\code> py -2          # Uruchomienie interaktywnej sesji z najnowszą zainstalowaną wersją 2.x
```

```
C:\code> py -3.1        # Uruchomienie interaktywnej sesji tylko z wersją 3.1
```

```
C:\code> py            # Uruchomienie domyślnej wersji (początkowo 2.x – patrz dalej)
```

Jeżeli zostanie użyty wiersz `#!` w pliku i podany numer wersji w parametrze polecenia, wtedy polecenie ma pierwszeństwo przed wierszem:

```
#! python3.2
```

```
...
```

```
...skrypt 3.x
```

```
...
```

```
C\code> py script.py      # Uruchomienie z wersją 3.2, zgodnie z dyrektywą w pliku
```

```
C\code> py -3.1 script.py  # Uruchomienie z wersją 3.1, nawet jeżeli jest zainstalowana wersja 3.2
```

Jeżeli wersja Pythona nie zostanie podana lub będzie określona tylko ogólnie, wtedy do wybrania konkretnej wersji są stosowane techniki heurystyczne. Na przykład, jeżeli zostanie wskazana wersja 2, wtedy wybierana jest najnowsza zainstalowana wersja 2.x. Wersja ta jest również preferowana, jeżeli skrypt, który został kliknięty lub którego nazwa została wpisana w wierszu polecen (np. `py m.py` lub `m.py`), nie zawiera wiersza `#!`, chyba że za pomocą zmiennej środowiskowej `PYTHON` zostanie określona domyślna wersja 3.x lub w pliku konfiguracyjnym umieszczony będzie odpowiedni wpis (więcej na ten temat za chwilę).

W dzisiejszym świecie podwójnego Pythona 2.x/3.x możliwość jawnego wskazywania wersji okazuje się przydatną funkcjonalnością systemu Windows, który wielu (a prawdopodobnie większość) początkujących programistów wykorzystuje do poznania tego języka. Opisane techniki, choć nie są wolne od potencjalnych pułapek, np. błędnie wpisanego wiersza `#!` lub

niewłaściwie wybranej domyślnej wersji 2.x, umożliwiają bezkonfliktowe uruchamianie w tym samym systemie plików przeznaczonych dla wersji 2.x i 3.x, jak również stanowią racjonalne rozwiązanie umożliwiające kontrolowanie wersji za pomocą wiersza poleceń.

Pełne informacje o programie uruchomieniowym, w tym bardziej zaawansowane funkcjonalności i przykłady użycia, które tutaj zostały opisane w skondensowanej formie lub pominięte, zawarte są w uwagach do poszczególnych wersji Pythona i dokumentacji PEP (w proponowanej wersji, do znalezienia za pomocą wyszukiwarki). Program umożliwia m.in. wybieranie wersji 32- lub 64-bitowej, określanie domyślnych ustawień w pliku konfiguracyjnym i definiowanie własnych rozszerzeń wiersza #!.

Podręcznik do programu uruchomieniowego

Dla czytelników znających język skryptowy systemu Unix opis zawarty w poprzednim podrozdziale jest wystarczający, aby mogli zacząć korzystać z programu uruchomieniowego. Inni znajdą w tej części dodatkowe informacje przedstawione w formie podręcznika z gotowymi przykładami użycia programu do samodzielnego przeanalizowania. Opisane są również dodatkowe szczegóły, więc nawet zaprawieni w bojach weterani Uniksa znajdą tu coś dla siebie, zanim przeniosą wszystkie swoje skrypty do systemu Windows.

Na początek przygotujmy prosty skrypt o nazwie *what.py*, który po uruchomieniu z wersją 2.x lub 3.x będzie wyświetlał wersję użytego interpretera Pythona. Wykorzystamy tu stałą `sys.version` zawierającą ciąg znaków, którego pierwsza część oddzielona spacjami od innych zawiera numer wersji:

```
#!python3
import sys
print(sys.version.split()[0])      # Wybranie pierwszej części ciągu
```

Aby samodzielnie wykonywać przykłady, należy powyższy kod wpisać do ulubionego edytora tekstu, zapisać go, następnie otworzyć wiersz poleceń i przejść za pomocą polecenia cd do katalogu, w którym został zapisany skrypt (tutaj jest to *C:\code*, ale może być dowolny inny — więcej wskazówek na ten temat jest zawartych w rozdziale 3.).

Pierwszy wiersz w powyższym skrypcie wskazuje wymaganą wersję Pythona. Na początku, zgodnie z przyjętą w systemie Unix konwencją, muszą znajdować się znaki #!, po nich nieobowiązkowa spacja i na końcu słowo `python3`. Na swoim komputerze mam zainstalowane wersje Pythona 2.7, 3.1, 3.2 i 3.3. Sprawdźmy teraz, jakie wersje będą wybrane, gdy ten wiersz będziemy zmieniać w opisany niżej sposób, stosować różne dyrektywy, parametry poleceń i domyślne ustawienia.

Krok 1: dyrektywa wersji w pliku

Gdy skrypt w opisanej wyżej formie zostanie uruchomiony, czy to przez kliknięcie jego ikony, czy wpisanie nazwy w wierszu poleceń, pierwszy wiersz zleci zarejestrowanemu programowi uruchomieniowemu użycie najnowszej zainstalowanej wersji 3.x:

```
#! python3
import sys
print(sys.version.split()[0])
```

```
C:\code> what.py          # Wybranie wersji wskazanej w dyrektywie  
3.3.0  
C:\code> py what.py      # Tak samo: wybranie najnowszej wersji 3.x  
3.3.0
```

Przypomnę, że spacja po znakach `#!` jest opcjonalna. Umieściłem ją tutaj, aby podkreślić ten fakt. Należy zwrócić uwagę, że pierwsze z powyższych poleceń jest równoważne zarówno kliknięciu ikony, jak i wpisaniu pełnego polecenia `py what.py`, dlatego że program `py.exe` został podczas instalacji skojarzony z plikami typu `.py`, tj. w rejestrze Windows został umieszczony odpowiedni wpis i program ten jest uruchamiany automatycznie.

Warto pamiętać, że w dokumentacji do programu uruchomieniowego (jak również w tym dodatku) termin *najnowsza* wersja oznacza wersję o *największym* numerze, przy czym dotyczy to najnowszej udostępnionej wersji, a nie zainstalowanej ostatnio na komputerze (np. jeżeli najpierw zostanie zainstalowana wersja 3.3, a potem 3.1, to wiersz `#! python3` spowoduje wybranie pierwszej z nich). Program uruchomieniowy przegląda wersje zainstalowane w systemie i wybiera tę o największym numerze, zgodnym ze wskazaną lub domyślną wersją. Jest to inne działanie niż w opisany wcześniej modelu, w którym zwycięża ostatnia zainstalowana wersja.

Jeżeli teraz w pierwszym wierszu umieścimy słowo `python2`, zostanie wybrana najnowsza (tj. o największym numerze) zainstalowana wersja 2.x. Poniżej pokazano, jak to zrobić. Pominięte zostały dwa ostatnie wiersze, ponieważ nie uległy zmianie.

```
#! python2  
...pozostała część skryptu bez zmian  
C:\code> what.py          # Wybranie najnowszej wersji 2.x zgodnie z dyrektywą  
#!  
2.7.3
```

W razie potrzeby można wersję określić dokładniej. Na przykład, jeżeli nie chcemy używać najnowszej wersji, musimy wpisać:

```
#! python3.1  
...  
C:\code> what.py          # Wybranie wersji 3.1 zgodnie z dyrektywą #!
```

3.1.4

Zasada ta obowiązuje nawet wtedy, gdy żądana wersja *nie jest zainstalowana*. Taki zapis jest wtedy traktowany przez program uruchomieniowy jako błąd:

```
#! python2.6  
...  
C:\code> what.py  
Requested Python version (2.6) is not installed
```

Nierozpoznany w systemie Unix wiersz `#!` również spowoduje wyświetlenie komunikatu o błędzie, chyba że w wierszu poleceń użyje się poprawnego parametru (co opisuje dokładnie następny podrozdział, a podrozdział o problemach z programem uruchomieniowym traktuje jako pułapkę):

```
#!/bin/python
```

```
...
C:\code> what.py
Unable to create process using '/bin/python "C:\code\what.py" '
C:\code> py what.py
Unable to create process using '/bin/python what.py'
C:\code> py -3 what.py
```

3.3.0

Program uruchomieniowy poprawnie rozpoznaje wiersz `#!`, jeżeli jest zapisany w jednym z czterech poniższych formatów:

```
#!/usr/bin/env python*
#!/usr/bin/python*
#!/usr/local/bin/python*
#!python*
```

Każdy wiersz rozpoczynający się od znaków `#!`, ale niezgodny z żadnym z powyższych formatów jest traktowany jako pełne polecenie uruchamiające proces lub plik i przekazywany w niezmienionej postaci do systemu Windows. W efekcie, jeżeli polecenie nie jest poprawne, pojawia się komunikat o błędzie, który widzieliśmy wcześniej. (Program uruchomieniowy obsługuje również niestandardowe rozszerzenia poleceń zdefiniowane w pliku konfiguracyjnym, które próbuje wykonać, zanim przekaże je systemowi Windows jako nieroznajmiane; tym przypadkiem nie będziemy się jednak tutaj zajmować).

W poprawnie sformułowanym wierszu `#!` ścieżka jest zapisywana zgodnie z konwencją przyjętą w systemie Unix, aby zapewnić uniwersalność kodu. Znak `*` umieszczony na końcu każdego z czterech pokazanych wyżej wierszy oznacza opcjonalny numer wersji Pythona, który można zapisać w jednym z trzech poniższych formatów:

Częściowym (np. `python3`)

Ten format powoduje wybranie wersji o największym dodatkowym numerze spośród innych o takim samym numerze głównym.

Pełnym (np. `python3.1`)

Wybranie wskazanej wersji. Można dodać opcjonalny sufiks `-32` oznaczający preferowaną wersję 32-bitową (np. `python3.1-32`).

Bez wersji (np. `python`)

Wybranie domyślnej wersji, tj. nr 2, jeżeli nie została skonfigurowana inna (np. przez nadanie zmiennej środowiskowej `PYTHON` wartości 3). Tu kryje się kolejna pułapka opisana w dalszej części rozdziału.

Plik, w którym nie ma wiersza `#!`, jest traktowany tak, jakby zawierał wiersz z ogólną nazwą `python`, tj. był zapisany w wymienionym wyżej formacie bez wersji. W takim przypadku uwzględniana jest wartość zmiennej środowiskowej `PYTHON`. Podobnie uwzględniane są zmienne środowiskowe w przypadku pierwszego, częściowego formatu. Na przykład, jeżeli w pliku zostanie umieszczone słowo `python3`, a zmienna `PYTHON` będzie miała wartość 3.1, wtedy zostanie wybrana wersja 3.1. Analogicznie, jeżeli plik będzie zawierał słowo `python2`, a zmienna `PYTHON2` będzie miała wartość 2.6, wtedy zostanie wybrana wersja 2.6. Domyślne ustawienia są opisane w dalszej części podręcznika.

Należy zwrócić uwagę, że wszystko, co w wierszu `#!` znajduje się po znaku `*`, jest traktowane jako parametry samego Pythona (tj. programu `python.exe`), chyba że używa się polecenia `py` z parametrami, które mają w takim wypadku pierwszeństwo:

```
#!/python3 [argumenty programu python.exe]  
...
```

Można stosować wszystkie argumenty opisane w dodatku A. W ten sposób zbliżyliśmy się do tematu parametrów programu uruchomieniowego stosowanych w wierszu poleceń, opisanych w następującym podrozdziale.

Krok 2: parametry w wierszu poleceń

Jak wspomniałem, jeżeli numer wersji Pythona nie zostanie określony w pliku, można go wskazać za pomocą parametru w wierszu poleceń. W tym celu należy użyć polecenia `py` lub `pyw` z parametrem i nie zdawać się na skojarzenia typów plików zapisane w rejestrze ani numer wersji wskazany w wierszu `#!`. W poniższym przykładzie użyty jest zmodyfikowany skrypt, który nie ma wiersza `#!`:

```
# Brak dyrektywy z wersją  
...  
C:\code> py -3 what.py      # Uruchomienie z wersją podaną w parametrze  
3.3.0  
C:\code> py -2 what.py      # Tak samo: uruchomienie z najnowszą zainstalowaną  
wersją 2.x  
2.7.3  
C:\code> py -3.2 what.py    # Tak samo: wyłącznie z wersją 3.2  
3.2.3  
C:\code> py what.py        # Uruchomienie z domyślną wersją (patrz opis  
niżej)  
2.7.3
```

Parametry polecenia mają pierwszeństwo przed numerem wersji określonym w dyrektywie w skrypcie:

```
#! python3.1  
...  
C:\code> what.py            # Uruchomienie z wersją podaną w dyrektywie  
3.1.4  
C:\code> py what.py         # Tak samo  
3.1.4  
C:\code> py -3.2 what.py    # Parametr ma pierwszeństwo przed dyrektywą  
3.2.3  
C:\code> py -2 what.py      # Tak samo  
2.7.3
```

Program uruchomieniowy akceptuje następujące parametry, które wiernie odpowiadają opisanej w poprzednim podrozdziale części oznaczonej symbolem *, umieszczonej na końcu pliku #!:

- -2: uruchomienie z najnowszą wersją 2.x,
- -3: uruchomienie z najnowszą wersją 3.x,
- -x.y: uruchomienie ze wskazaną wersją (x oznacza cyfrę 2 lub 3),
- -x.y-32: uruchomienie ze wskazaną wersją 32-bitową.

Polecenie programu uruchomieniowego ma następujący ogólny format:

```
py [py.exe parametry] [python.exe parametry] script.py [script.py parametry]
```

Wszystkie znaki wpisane po parametrach programu uruchomieniowego (jeżeli zostały użyte) są przekazywane do programu *python.exe*. Zazwyczaj są to argumenty samego Pythona, nazwa skryptu i jego parametry.

Z poleceniem py można również stosować typowe parametry -m mod, -c cmd i - oraz inne, opisane w dodatku A. Jak wspomniałem wcześniej, na końcu wiersza #! też można umieszczać parametry programu *python.exe*, niemniej parametry podane w wierszu poleceń mają wtedy pierwszeństwo.

Aby zobaczyć powyższą zasadę w działaniu, utwórzmy nowy skrypt, który dodatkowo będzie wyświetlał parametry podane w wierszu polecień. Stała `sys.argv` zawiera parametry skryptu. W tym przykładzie parametr -i programu *python.exe* powoduje przejście do trybu interaktywnego (>>>) po wykonaniu skryptu:

```
# Skrypt args.py wyświetlający również parametry
import sys
print(sys.version.split()[0])
print(sys.argv)
C:\code> py -3 -i args.py -a 1 -b -c # -3: py, -i: python, reszta: skrypt
3.3.0
['args.py', '-a', '1', '-b', '-c']
>>> ^Z
C:\code> py -i args.py -a 1 -b -c      # Parametry Pythona i skryptu
2.7.3
['args.py', '-a', '1', '-b', '-c']
>>> ^Z
C:\code> py -3 -c print(99)           # -3: py, reszta: python "-c cmd"
99
C:\code> py -2 -c "print 99"
99
```

Należy zwrócić uwagę, że pierwsze dwa polecenia uruchamiają domyślne wersje Pythona, ponieważ w skrypcie nie ma wiersza #!. Poniekąd przypadkowo wprowadza nas to w temat następnego podrozdziału.

Krok 3: stosowanie i zmianianie ustawień domyślnych

Jak wspomniałem, program uruchomieniowy domyślnie wybiera wersję 2.x, jeżeli w wierszu `#!` jest użyte ogólne polecenie `python` bez konkretnego numeru wersji. Zasada ta obowiązuje niezależnie od tego, czy wpisana jest pełna ścieżka (np. `#!/usr/bin/python`), czy nie (`#!python`). Poniżej zilustrowany jest ten drugi przypadek na przykładzie oryginalnego skryptu `what.py`:

```
#!python
...
C:\code> what.py          # Uruchomienie z domyślnymi ustawieniami
2.7.3
```

Domyślne ustawienia są stosowane również wtedy, gdy dyrektywy w ogóle nie ma. Jest to prawdopodobnie najczęstszy przypadek, gdy kod jest przeznaczony wyłącznie do uruchamiania w systemie Windows:

```
# Brak dyrektywy programu uruchomieniowego
...
C:\code> what.py          # Uruchomienie z domyślnymi ustawieniami
2.7.3
C:\code> py what.py       # Tak samo
2.7.3
```

Jednak za pomocą pliku inicjującego lub zmiennej środowiskowej można skonfigurować program uruchomieniowy tak, aby domyślnie wybierał wersję 3.x. Dotyczy to uruchamiania skryptów zarówno za pomocą wiersza poleceń, jak i ikon (w wyniku skojarzenia typów plików z programami `py.exe` i `pyw.exe` w rejestrze Windows):

```
# Brak dyrektywy programu uruchomieniowego
...
C:\code> what.py          # Uruchomienie z domyślnymi ustawieniami
2.7.3
C:\code> set PYTHON=3      # Lub poprzez Panel sterowania / System
C:\code> what.py          # Uruchomienie z domyślnymi, zmienionymi
                           ustawieniami
3.3.0
```

Jak sugerowałem wcześniej, aby kontrolować wersje, można również modyfikować odpowiednie zmienne środowiskowe. W ten sposób uściąła się wybór konkretnej wersji bez zdawania się na zainstalowaną wersję o największym numerze dodatkowym:

```
#!python3
...
C:\code> py what.py        # Uruchomienie z "najnowszą" wersją 3.x
3.3.0
```

```
C:\code> set PY_Python3=3.1      # Użycie zmiennej PY_Python2 w przypadku wersji  
2.x  
C:\code> py what.py             # Zastąpienie wersji o największym numerze  
dodatkowym
```

3.1.4

Zmiany wprowadzone za pomocą polecenia `set` obowiązują tylko w bieżącym oknie wiersza poleceń. Aby zmienić ustawienia globalnie, należy użyć *Panelu sterowania* i opcji *System* (dodatkowe informacje na ten temat znajdują się w dodatku A). W zależności od tego, z jaką wersją ma być uruchamiana większość skryptów, powyższe ustawienia mogą, ale nie muszą być stosowane. Wielu użytkownikom wersji 2.x prawdopodobnie wystarczą domyślne ustawienia, które w razie potrzeby mogą zmieniać za pomocą wiersza `#!` lub parametru polecenia `py`.

Jednak w przypadku skryptów, które nie zawierają dyrektywy, bardzo ważna jest wartość zmiennej środowiskowej `PYTHON`. Większość programistów, którzy w przeszłości używali Pythona w systemie Windows, zapewne oczekuje, że po zainstalowaniu wersji 3.3 będzie ona stosowana domyślnie (szczególnie jeżeli jest to pierwsza instalacja). Jest to pozorny paradoks, który wprowadza nas do następnego podrozdziału.

Pułapki nowego programu uruchomieniowego

Nowy program uruchomieniowy dostarczany z wersją 3.3 jest miłym dodatkiem i szkoda, że wraz z innymi udoskonaleniami nie pojawił się wcześniej. Niestety, cechuje go pewna niekompatybilność z poprzednimi wersjami, co może być pewnym utrudnieniem w dzisiejszym wielowersyjnym świecie Pythona, a nawet uniemożliwiać działanie niektórych istniejących programów. Dotyczy to przykładów opisanych w tej książce i wielu innych skryptów. Podczas dostosowywania kodów do wersji 3.3 napotkałem trzy problemy z programem uruchomieniowym, o których warto wspomnieć:

- nierozniany w systemie Unix wiersz `#!` powoduje, że skrypt w systemie Windows nie działa;
- program uruchomieniowy domyślnie wybiera wersję 2.x, jeżeli nie wskaże się innej;
- wartość zmiennej `PATH` nie jest uwzględniana, co wydaje się niezrozumiałe.

Pozostała część tego podrozdziału zawiera omówienie powyższych trzech problemów. Do zilustrowania niekompatybilności programu uruchomieniowego wykorzystane są przykłady z czwartego wydania książki *Programming Python*, ponieważ dostosowanie skryptów napisanych w wersji 3.1/3.2 do wersji 3.3 jest moim pierwszym doświadczeniem z nowym programem uruchomieniowym. W tym konkretnym przypadku po zainstalowaniu wersji 3.3 wiele przykładów z powyższej książki przygotowanych w wersjach 3.1 i 3.2 przestało działać. Opisane tu przyczyny mogą również zakłócać działanie skryptów napisanych przez czytelników.

Pułapka 1: nierozniany w Uniksie wiersz `#!` uniemożliwia uruchomienie skryptu

Nowy program uruchomieniowy, w przeciwieństwie do innej często stosowanej w Uniksie formy `#!/bin/env python` (obowiązkowej w niektórych systemach), rozpoznaje wiersze rozpoczynające się od frazy `#!/usr/bin/env python`. Skrypty, w których wykorzystywana jest ta druga forma, w tym przykłady z mojej książki, w przeszłości działały poprawnie w systemie Windows, ponieważ wiersze `#!` były traktowane jak komentarze i pomijane. Te same skrypty nie

działają w wersji 3.3, ponieważ nowy program uruchomieniowy nie rozpoznaje formatu dyrektywy i wyświetla komunikaty o błędzie.

Uogólniając, skrypt zawierający dowolny nierożpoznany w Uniksie wiersz `#!` nie działa w systemie Windows. Dotyczy to każdego skryptu, w którym wiersz `#!` zawiera inną frazę niż jedna z czterech opisanych wcześniej, tj. `/usr/bin/env python*`, `/usr/bin/python*`, `/usr/local/bin/python*` i `python*`. Każda inna fraza uniemożliwia uruchomienie skryptu i wymaga wprowadzenia zmian w kodzie. Na przykład powszechnie stosowana dyrektywa `#!/bin/python` również powoduje, że skrypt ulega awarii w systemie Windows, chyba że w wierszu poleceń wskaże się konkretną wersję Pythona.

Programy przeznaczone tylko dla systemu Windows zazwyczaj w ogóle nie zawierają wiersza `#!`, jednak dyrektywa ta jest często stosowana w skryptach przeznaczonych dodatkowo dla systemu Unix. Traktowanie nierożpoznanych w systemie Unix wierszy jako błędnych w systemie Windows wydaje się radykalnym pomysłem, tym bardziej że jest to zaskakująca zmiana wprowadzona w wersji 3.3. Dlaczego nierożpoznany wiersz `#!` nie jest po prostu ignorowany i nie jest przyjmowana domyślna wersja Pythona tak jak we wszystkich dotychczasowych wersjach w systemie Windows? Być może zostanie to poprawione w jednej z kolejnych wersji 3.x (istnieje już pewna presja), ale dzisiaj trzeba zmieniać wszystkie pliki zawierające wiersze `#!/bin/env` lub inne nierożpoznane frazy, jeżeli mają być poprawnie obsługiwane przez program uruchomieniowy w wersji 3.3 w systemie Windows.

Wpływ zmian na przykłady użyte w książce i korekta

Kilkadziesiąt wykorzystanych w tej książce przykładowych skryptów rozpoczynających się od wierszy `#!` po przeniesieniu do wersji 3.3 przestało działać. Niestety, dotyczy to również kilku demonstracyjnych, przyjaznych dla użytkownika skryptów, m.in. *PyGadgets* i *PyDemos*. Aby je poprawić, musiałem wpisać w nich akceptowną frazę `#!/usr/bin/env python`. Innym rozwiązaniem może być zmiana skojarzeń typów plików (na przykład pliki `.py` można skojarzyć z programem `python.exe` zamiast `py.exe`) i całkowita rezygnacja z programu uruchomieniowego. Jednak takie podejście kwestionuje zalety programu uruchomieniowego i dla użytkowników, szczególnie początkujących, może być niezrozumiałe.

Pojawia się jeszcze jeden problem: użycie parametru w wierszu poleceń, nawet parametru programu `python.exe`, niweluje opisany wyżej efekt i skutkuje przyjęciem domyślnych ustawień. Polecenia `m.py` i `py m.py` powodują wyświetlenie komunikatu o nierożpoznanym wierszu `#!`, natomiast polecenie `py -i m.py` uruchamia skrypt z domyślnymi ustawieniami Pythona. Wygląda to na błąd programu uruchomieniowego, ale też dotyczy domyślnych ustawień, o których mowa w następnym podrozdziale.

Pułapka 2: domyślna wersja 2.x w programie uruchomieniowym

Ciekawe, że program uruchomieniowy w wersji 3.3 domyślnie wybiera wersję 2.x, jeżeli jest zainstalowana, ale w uruchamianym skrypcie wersja nie jest jawnie określona. Oznacza to, że skrypt, w którym nie ma wiersza `#!` lub wiersz ten jest i zawiera ogólne polecenie `python`, jest domyślnie uruchamiany z wersją 2.x po kliknięciu jego ikony, wpisaniu nazwy w wierszu poleceń (`m.py`) lub użyciu programu uruchomieniowego bez wskazania wersji (`py m.py`). Zasada ta obowiązuje nawet, jeżeli po wersji 2.x zostanie zainstalowana wersja 3.3. Z tego powodu wiele skryptów przystosowanych do wersji 3.x może nie działać poprawnie.

Konsekwencje takiego stanu rzeczy mogą być dość istotne. Na przykład po zainstalowaniu wersji 3.x skrypt bez dyrektywy uruchomiony przez kliknięcie ikony może ulec awarii, ponieważ skojarzony z nim program uruchomieniowy wybierze domyślną wersję 2.x. Na pewno nie będzie to miłe doświadczenie dla początkującego użytkownika Pythona! Przyjęte jest założenie, że skrypt w wersji 3.x nie ma dyrektywy `#!` zawierającej jawnie polecenie `python3`, ale większość skryptów przeznaczonych dla systemu Windows nie ma tej dyrektywy w ogóle. Zatem wiele

skryptów utworzonych przed pojawieniem się nowego programu uruchomieniowego nie będzie spełniało powyższego założenia. Większość użytkowników wersji 3.x będzie musiała po zainstalowaniu Pythona odpowiednio ustawić zmienną środowiskową PY_Python, co nie jest szczególnie wygodnym rozwiązaniem.

Brak jawnego wskazania numeru wersji może skutkować niejednoznacznym uruchamianiem skryptu również w systemie Unix. Dlatego programiści często wykorzystują symboliczne łącza wiążące program python z określona wersją (dzisiaj jest to zazwyczaj 2.x, tak jak to emuluje program uruchomieniowy w systemie Windows). Ale tak jak w przypadku poprzedniej pułapki: skrypty, które wcześniej działały poprawnie, nie powinny po uruchomieniu w systemie Windows z wersją 3.3 generować nowych błędów. Większość programistów nie spodziewa się, że komentarze w systemie Unix będą miały znaczenie w systemie Windows i że zaraz po zainstalowaniu wersji 3.3 będzie domyślnie stosowana wersja 2.x.

Wpływ zmian na przykłady w książce i korekta

Domyślne wybieranie wersji 2.x spowodowało, że wiele skryptów napisanych w wersji 3.x, które nie miały dyrektywy `#!` albo miały właściwą dla systemu Unix dyrektywę `#!/usr/bin/python`, przestało działać po zainstalowaniu wersji 3.3. Aby rozwiązać drugi problem, należy wszystkie tego rodzaju skrypty zmienić tak, aby zamiast polecenia `python` zawierały `python3`. W celu rozwiązania obu problemów w jednym kroku należy zmienić globalne ustawienia programu uruchomieniowego, tak aby domyślnie wybierał wersję 3.x. Można to zrobić, odpowiednio zmieniając plik konfiguracyjny `py.ini` (szczegóły są opisane w dokumentacji programu uruchomieniowego) lub ustawiając zmienną środowiskową PY_Python w sposób pokazany w poprzednich przykładach (np. `set PY_Python=3`). Jak pisałem wcześniej, innym rozwiązaniem jest ręczna zmiana skojarzeń typów plików. Jednak żadna z tych opcji nie jest tak prosta jak obowiązująca we wcześniejszych wersjach.

Pułapka 3: nowa opcja modyfikacji zmiennej PATH

Program instalacyjny Pythona 3.3 nie tylko umieszcza program uruchomieniowy w systemie operacyjnym, ale też może automatycznie dodawać do zmiennej środowiskowej PATH katalog zawierający plik `python.exe`. Wynika to z chęci ułatwienia pracy początkującym programistom używającym systemu Windows — wystarczy, że zamiast pełnej ścieżki będą wpisywać polecenie `python`. Nie jest to funkcjonalność samego programu uruchomieniowego, więc nie zakłóca działania skryptów i w żaden sposób nie wpływa na działanie przykładów opisanych w tej książce. Koliduje jednak z działaniem i przeznaczeniem programu uruchomieniowego, dlatego najlepiej jest z niej zrezygnować. Jest to dość subtelna kwestia, którą teraz wyjaśnię.

Jak pisałem wcześniej, nowe pliki wykonywalne `py.exe` i `pyw.exe` są domyślnie instalowane w jednym z przeszukiwanych katalogów systemowych. Zatem w celu ich uruchomienia nie trzeba wpisywać pełnej ścieżki ani modyfikować zmiennej PATH. Jeżeli skrypty uruchamia się za pomocą polecenia `py`, a nie `python`, nowa wartość zmiennej PATH nie ma znaczenia. W praktyce w większości przypadków polecenie `py całkowicie zastępuje` polecenie `python`. Zważywszy, że w wyniku skojarzenia typów plików uruchamiane jest polecenie `py` lub `pyw`, a nie `python`, użytkownicy też powinni używać tych poleceń. Stosowanie polecenia `python` zamiast `py` może okazać się zbędne i niespójne, a nawet skutkować wybieraniem innych wersji niż stosowane przez program uruchomieniowy, szczególnie gdy oba schematy będą się od siebie różnić. Podsumowując, dodanie do zmiennej PATH katalogu zawierającego program `python` stoi w sprzeczności z przeznaczeniem nowego programu uruchomieniowego i jest podatne na błędy.

Należy również zwrócić uwagę, że modyfikując zmenną PATH, zakładamy, że polecenie `python` uruchamia wersję 3.3. Jednak ta funkcjonalność jest domyślnie *wyłączana* i jeżeli chcemy z niej korzystać, należy ją zaznaczyć w programie instalacyjnym (lub pozostawić niezaznaczoną w przeciwnym wypadku). Z powodu opisanej wcześniej pułapki nr 2 wielu użytkowników będzie musiało nadać zmiennej PY_Python wartość 3, aby uruchamiać skrypty poprzez klikanie ikon, co nie wydaje się prostszym rozwiązaniem od modyfikacji zmiennej PATH — kroku

eliminowanego przez nowy program uruchomieniowy. Lepszym wyjściem jest korzystanie wyłącznie z programu uruchomieniowego i modyfikacja zmiennej PYTHON w razie potrzeby.

Podsumowanie: ostateczny wynik dla systemu Windows

Trzeba przyznać, że opisane tutaj pułapki są nieuniknioną konsekwencją próby ujednolicenia obsługi systemów Unix i Windows, w których zainstalowanych jest kilka wersji Pythona. Jednak w zamian można w spójny sposób zarządzać skryptami i instalacjami różnych wersji. Może się okazać, że program uruchomieniowy, dostarczany od wersji 3.3, będzie jej głównym atutem, gdy użytkownicy zaczną go używać i przezwyciężą początkowe trudności wynikające z niekompatybilności różnych wersji.

W praktyce warto również nabrać nawyku wpisywania w skryptach w systemie Windows wierszy `#!`, które są kompatybilne z Unixem i jawnie wskazują wersję (np. `#!/usr/bin/python3`). W ten sposób nie tylko deklaruje się wymagania kodu i zapewnia jego poprawne działanie w systemie Windows, ale też omija domyślne ustawienia programu uruchomieniowego i daje możliwość uruchamiania skryptu w systemie Unix w przyszłości.

Należy jednak pamiętać, że program uruchomieniowy może zakłócać działanie skryptów, które zawierają wiersze `#!` i dotychczas funkcjonowały poprawnie. Program może na przykład wybrać domyślną wersję, nieodpowiednią do uruchomienia skryptu, przez co niezbędne będzie wprowadzenie zmian w konfiguracji i kodzie, czego pierwotnie chcieliśmy uniknąć. Nowy szef jest lepszy niż stary, ale wygląda na to, że obaj chodzili do tej samej szkoły.

Więcej informacji o używaniu Pythona w systemie Windows jest dostępnych w dodatku A (instancja i konfiguracja), rozdziale 3. (ogólne pojęcia) i w dokumentacji właściwej dla wybranego systemu operacyjnego.

Dodatek C Zmiany w języku Python a niniejsza książka

Niniejszy dodatek stanowi krótkie podsumowanie zmian wprowadzonych w ostatnich wersjach Pythona i opisanych w kolejnych wydaniach tej książki. Zawiera również odniesienia do zawartych w książce opisów różnych funkcjonalności. Dodatek jest przeznaczony zarówno dla czytelników wcześniejszych wydań, jak również programistów aktualizujących kody napisane w poprzednich wersjach.

Poniżej wymienione są zmiany wprowadzone w języku Python, opisane w kolejnych wydaniach książki:

- niniejsze wydanie, z 2013 r., opisuje wersje Pythona 3.3 i 2.7;
- czwarte wydanie, z 2009 r., opisuje wersje 2.6 i 3.0 (oraz niektóre funkcjonalności z wersji 3.1);
- trzecie wydanie, z 2007 r., opisuje wersję 2.5;
- pierwsze i drugie wydanie, z lat 1999 i 2003, opisują wersje 2.0 i 2.2;
- pierwowzór z 1996 r., książka *Programming Python*, opisuje wersję 1.3.

Zatem piąte wydanie zawiera wymienione niżej zmiany wprowadzone w wersjach 2.7, 3.2 i 3.3. Zmiany wprowadzone w wersjach 2.6, 3.0 i 3.1 zostały zawarte w wydaniach *czwartym* i *piątym*. Zaprezentowane zostały również krótko zmiany opisane w trzecim wydaniu, które dzisiaj mają jedynie historyczne znaczenie.

Należy zwrócić uwagę, że niniejszy dodatek skupia się na najważniejszych zmianach i nie stanowi pełnego przewodnika po ewolucji Pythona. Pełniejszy opis zmian wprowadzonych w poszczególnych wersjach jest zawarty w rozdziale *What's New* („Co nowego?”) w standardowej dokumentacji dostępnej na stronie python.org, w sekcji *Download*. Dokumentacja ta i zestaw podręczników zostały opisane w rozdziale 15.

Najważniejsze różnice między wersjami 2.x i 3.x

Niniejszy dodatek w znacznej części stanowi podsumowanie zmian wprowadzonych w Pythonie i opisanych w poszczególnych rozdziałach książki. Czytelnikom, którzy poszukują informacji o najważniejszych różnicach pomiędzy wersjami 2.x a 3.x, powinien wystarczyć ten podrozdział. Należy zwrócić uwagę, że są tu porównane ostatnie wersje, tj. 2.7 i 3.3. Wiele funkcjonalności dostępnych w wersji 3.x nie zostało tu opisanych, ponieważ albo zostały wprowadzone w wersji 2.6 (np. instrukcja `width` i dekoratory klas), albo dostosowane wstępnie do wersji 2.7 (np. wyrażenia zbiorowe i słownikowe), choć nie ma ich we wcześniejszych wersjach 2.x. W kolejnych podrozdziałach zawarte są bardziej szczegółowe informacje o zmianach wprowadzonych w poszczególnych wersjach, natomiast w przyszłości takich informacji należy szukać w dokumentacji Pythona w części *What's New*.

Zmiany w wersji 3.x

Poniżej wymienione są narzędzia, które zmieniały się w kolejnych wersjach Pythona:

- *Model Unicode ciągu znaków.* W wersji 3.x zwykły typ `str` obsługuje wszystkie znaki Unicode, w tym również ASCII, a osobny typ `byte` reprezentuje surowe sekwencje bajtów. W wersji 2.x typ `str` obsługuje 8-bitowe znaki (w tym ASCII), a osobny typ `unicode` opcjonalnie obsługuje bogatszy tekst Unicode.
- *Model plików.* W wersji 3.x instrukcja `open` tworzy pliki, których typy zależą od zawartości. Pliki tekstowe są kodowane według standardu Unicode, a ich treść jest reprezentowana przez typ `str`. Natomiast zawartość plików binarnych reprezentuje typ `bytes`. W wersji 2.x do obsługi plików wykorzystywane są różne interfejsy. Pliki tworzone za pomocą metody `open` zawierają 8-bitowe znaki lub bajty danych reprezentowane przez typ `str`, natomiast metoda `codecs.open` implementuje kodowanie Unicode.
- *Model klasy.* W wersji 3.x wszystkie klasy pochodne od klasy `object` automatycznie przejmują zmiany i rozszerzenia klas w *nowym stylu*, w tym odmienny adres dziedziczenia, kierowanie wbudowanych operacji i algorytm przeszukiwania MRO drzewa nazw. W wersji 2.x zwykłe klasy są zgodne z *normalnym modelem*, natomiast jawne dziedziczenie klasy `object` lub innych wbudowanych typów opcjonalnie aktywuje model klas w nowym stylu.
- *Wbudowane elementy iterowalne.* W wersji 3.x typy `map`, `zip`, `range`, `filter`, klucze słowników, wartości i elementy są obiektami iterowalnymi, które generują wartości na żądanie. W wersji 2.x metody te tworzą rzeczywiste listy.
- *Wyświetlanie.* W wersji 3.x jest dostępna wbudowana funkcja, do której argumentów można odwoływać się za pomocą słów kluczowych, natomiast w wersji 2.x jest przeznaczona do tego celu instrukcja o specjalnej składni.
- *Względny import.* Zarówno w wersji 2.x, jak i 3.x dostępna jest instrukcja `from` i instrukcje do względnego importu, jednak w wersji 3.x zmienione zostały zasady wyszukiwania pakietów. Podczas normalnego importu pomijany jest własny katalog pakietu.
- *Dzielenie liczb.* Zarówno w wersji 2.x, jak i 3.x dostępny jest operator `//` zaokrąglający wynik w góre. Jednak w wersji 3.x operator `/` daje w wyniku liczbę rzeczywistą, natomiast w wersji 2.x typ wyniku zależy od typów dzielonych liczb.
- *Typ całkowity.* W wersji 3.x jest jeden typ całkowity o rozszerzonej precyzji. W wersji 2.x dostępny jest typ `int`, rozszerzony typ `long` oraz automatyczna konwersja typu danych na `long`.
- *Zakresy wyrażeń.* W wersji 3.x zmienne we wszystkich rodzajach wyrażeń — listowych, zbiorowych, słownikowych i generatorowych — są lokalne. W wersji 2.x w wyrażeniu listowym tak nie jest.
- *PyDoc.* Zgodny ze wszystkimi przeglądarkami interfejs otwierany poleceniem `pydoc -b` jest obsługiwany, począwszy od wersji 3.2, a w wersji 3.3 jest wymagany. W wersji 2.x zamiast niego można używać oryginalnego interfejsu graficznego otwieranego poleceniem `pydoc -g`.
- *Przechowywanie kodu bajtowego.* Począwszy od wersji 3.2, pliki z kodami bajtowymi są zapisywane w podkatalogu `_pycache_` katalogu źródłowego, a ich nazwy zawierają oznaczenia wersji. W wersji 2.x pliki z kodami bajtowymi są zapisywane w katalogu źródłowym i nadawane są im ogólne nazwy.
- *Wbudowane wyjątki systemowe.* Począwszy od wersji 3.3, zmieniła się hierarchia wyjątków systemowych i operacji wejścia/wyjścia. Pojawiły się dodatkowe, bardziej szczegółowe kategorie. W wersji 2.x w przypadku wystąpienia błędu systemowego trzeba czasami sprawdzać atrybuty wyjątku.
- *Porównywanie i sortowanie.* W wersji 3.x operacje porównywania wartości różnych typów oraz słowników są traktowane jak błędy, a operacje sortowania nie dotyczą mieszanych typów i ogólnych funkcji porównujących (zamiast tego należy stosować instrukcję `key`). W wersji 2.x dopuszczalne są wszystkie powyższe formy.
- *Wyjątki tekstowe i funkcje modułowe.* W wersji 2.6 znikły wyjątki oparte na ciągach znaków, nie ma ich również w wersji 3.x (zamiast nich należy stosować klasy). Funkcje

modułu `string` odpowiadające metodom ciągu znaków również zostały zlikwidowane w wersji 3.x.

- *Usunięte elementy języka.* Zgodnie z tabelą C.3 w wersji 3.x wiele elementów języka, m.in. `reload`, `apply`, `'x'`, `<>`, `0177`, `999L`, `dict.has_key`, `raw_input`, `xrange`, `file`, `reduce` i `file.readlines`, zostało usuniętych, zmienionych i przeniesionych w inne miejsca.

Rozszerzenia dostępne tylko w wersji 3.x

Poniższa lista podsumowuje narzędzia dostępne tylko w wersji 3.x:

- *Rozszerzone przypisania sekwencji.* W wersji 3.x można w operacji przypisania sekwencji stosować znak * powodujący umieszczenie w liście niedopasowanych iterowalnych elementów. W wersji 2.x podobny efekt można osiągnąć za pomocą wycinków.
- *Nazwy nielokalne.* W wersji 3.x jest dostępna instrukcja `nonlocal` umożliwiająca modyfikowanie lokalnych zmiennych danej funkcji wewnętrz zagnieżdżonych funkcji. W wersji 2.x podobny efekt można uzyskać za pomocą atrybutów funkcji, mutowalnych obiektów i stanów klasy.
- *Adnotacje funkcji.* W wersji 3.x można typy argumentów funkcji i ich wyników opatrywać adnotacjami, tj. obiekttami wykorzystywanyimi wewnątrz funkcji, i w żaden inny sposób. W wersji 2.x zazwyczaj podobny efekt można uzyskać za pomocą dodatkowych obiektów i argumentów dekoratora.
- *Argumenty ze słowami kluczowymi.* W wersji 3.x można definiować argumenty funkcji, którym wartości można przypisywać wyłącznie za pomocą słów kluczowych, zazwyczaj wykorzystywanych jako dodatkowe opcje konfiguracyjne. W wersji 2.x podobny efekt można uzyskać poprzez analizę argumentów i pobieranie danych ze słownika.
- *Łańcuchowanie wyjątków.* W wersji 3.x można wyjątki łączyć w łańcuchy i umieszczać je w komunikatach o błędach za pomocą rozszerzenia `raise from`. W wersji 3.3 stała `None` umożliwia przerwanie łańcucha.
- *Instrukcja yield from.* Począwszy od wersji 3.3, instrukcja `yield from` może delegować metodę do zagnieżdzonego generatora. W wersji 2.x w prostszych przypadkach można osiągnąć podobny efekt za pomocą pętli `for`.
- *Pakiety przestrzeni nazw.* W wersji 3.3 model pakietów został rozbudowany i pozwala tworzyć bez użycia pliku inicjującego pakiety obejmujące kilka katalogów. W wersji 2.x podobny efekt można uzyskać, importując rozszerzenia.
- *Program uruchomieniowy w systemie Windows.* Począwszy od wersji 3.3, Python jest oferowany wraz z programem uruchomieniowym, który jest również dostępny osobno dla innych wersji, w tym 2.x.
- *Szczegóły wewnętrzne.* Począwszy od wersji 3.2, wątki są implementowane jako wycinki czasowe, a nie wirtualne instrukcje maszynowe. Od wersji 3.3 tekst Unicode jest zapisywany z użyciem danych o zmiennej długości, a nie bajtów o stałej wielkości. W wersji 2.x model ciągów znaków generalnie minimalizuje użycie kodowania Unicode.

Ogólne uwagi do zmian w wersji 3.x

Choć wersja 3.x Pythona opisana w dwóch ostatnich wydaniach tej książki jest w większości tym samym językiem co wersja 2.x, jednak pomiędzy nimi są istotnie różnice. Zgodnie z opisem w przedmowie i podsumowaniem w poprzednim podrozdziale nieopcjonalny model Unicode, obowiązkowe klasy w nowym stylu, zwiększyły nacisk na generatory i inne funkcjonalne narzędzia stanowią zupełnie nową jakość.

Wersja 3.x jako całość jest czystszym, ale też pod wieloma względami bardziej skomplikowanym językiem, który opiera się na bardziej zaawansowanych pojęciach. Jego twórcy, wprowadzając niektóre zmiany, przyjęli założenie, że aby nauczyć się Pythona, trzeba znać Pythona. W

przedmowie wspomniałem o kilku ważniejszych cyklicznych zależnościach między pojęciami w wersji 3.x, implikujących kolejne zależności.

Weźmy pierwszy z brzegu przykład: powody opakowania widoków słowników w listę w wersji 3.x są niezwykle wyrafinowane i wymagają uprzedniego przyswojenia sobie dużej wiedzy, przynajmniej na temat widoków, generatorów i protokołu iteracyjnego. Podobnie argumenty ze słowami kluczowymi wymaganymi w niektórych narzędziach (np. wyświetlających dane, formatujących ciągi znaków, tworzących i sortujących słowniki) pojawiają się, jeszcze zanim poczatkujący programista dokładnie zrozumie działanie funkcji, w których te narzędzia są wykorzystywane. Dlatego jednym z celów tej książki jest wsparcie w uzupełnianiu luk w wiedzy w dzisiejszym świecie podwójnych wersji 2.x/3.x.

Zmiany w bibliotekach i narzędziach

W wersji 3.x Pythona zostało wprowadzonych wiele innych zmian, które nie zostały tutaj wymienione tylko dlatego, że nie odnoszą się do niniejszej książki. Niektóre standardowe biblioteki i narzędzia programistyczne nie dotyczą rdzenia języka. Inne (np. `timeit`) zostały albo wspomniane, albo były opisane we wszystkich wydaniach (np. *PyDoc*).

Aby opis był kompletny, w tym podrozdziale zostały wymienione udoskonalenia wprowadzone w wersji 3.x w bibliotekach i narzędziach. Niektóre zmiany w tych kategoriach są dodatkowo wymienione w dalszej części tego dodatku w odniesieniu do wydania książki i wersji Pythona, w których zostały wprowadzone.

Zmiany w standardowej bibliotece

Formalnie rzecz biorąc, standardowa biblioteka Pythona nie jest częścią rdzenia języka, który jest tematem tej książki, mimo że jest zawsze dostarczana z interpreterem i jest wykorzystywana we wszystkich programach. W rzeczywistości biblioteka nie była uwzględniona w moratorium na tymczasowe zmiany podczas opracowywania wersji 3.2.

Z tego powodu zmiany w standardowej bibliotece są o wiele dokładniej opisane w książkach poświęconych tworzeniu aplikacji, np. *Programming Python*, niż tutaj. Choć w wersji 3.x wciąż jest dostępna większość funkcjonalności standardowej biblioteki, jednak panuje tu większa swoboda w zmienianiu nazw modułów, grupowaniu ich w pakiety i modyfikowaniu interfejsu API.

Są też głębsze zmiany. Na przykład model *Unicode* wprowadził w wersji 3.x istotne zmiany w standardowej bibliotece, które mogą dotyczyć wszystkich programów operujących na plikach, nazwach plików, katalogach, strumieniach, deskryptorach plików, gniazdach sieciowych, interfejsach graficznych, protokołach internetowych (np. FTP, e-mail), skryptach CGI, wszelkiego rodzaju treściach WWW, niektórych bazach danych (np. plikach DBM), obiektach *shelve* i *pickle*.

Bardziej rozbudowana lista zmian wprowadzonych w standardowej bibliotece w wersji 3.x znajduje się w dokumencie *What's New* zawartym w zestawie podręczników do Pythona (szczególnie w wersji 3.0). Wspomniana wcześniej książka *Programming Python*, poświęcona głównie wersji 3.x, również może stanowić przewodnik po wprowadzonych w tej wersji zmianach.

Zmiany w narzędziach

Choć większość narzędzi programistycznych (np. do diagnozowania, profilowania, mierzenia czasu wykonania i testowania kodu) jest taka sama w wersjach 2.x i 3.x, to wraz z rozwojem języka i bibliotek zostało wprowadzonych kilka istotnych zmian. Między innymi z systemu do tworzenia dokumentacji modułów *PyDoc* został usunięty dotychczasowy interfejs graficzny i zastąpiony interfejsem przeglądarkowym.

Inne istotne zmiany wprowadzone w tej dziedzinie to: pakiet *distutils* służący do dystrybuowania i instalowania zewnętrznego oprogramowania został w wersji 3.x zastąpiony nowym systemem *pakietującym*; opisany w tej książce nowy schemat *_pycache_* przechowywania kodu bajtowego z racji wprowadzonych udoskonaleń potencjalnie może zmienić działanie wielu narzędzi i programów; dodatkowo zmieniona w wersji 3.2 wewnętrzna implementacja wątków zmniejszająca spętrzenia poprzez modyfikowanie globalnej blokady interpretera (GIL) wykorzystuje wycinki bezwzględnego czasu, a nie licznik instrukcji maszyny wirtualnej.

Migracja do wersji 3.x

Aby przejść od wersji 2.x do 3.x, warto wypróbować dostarczany razem z wersją 3.x skrypt *2to3* do automatycznej konwersji kodu. Obecnie jest on dostępny w folderze instalacyjnym *Tools\Scripts* oraz w internecie. Skrypt ten nie przekłada wszystkich elementów języka, głównie skupia się na jego rdzeniu. Dodatkowo może być inny interfejs API standardowej biblioteki. Niemniej jednak wykonuje dobrą robotę i pozwala dużą część kodu napisanego w wersji 2.x przenieść do wersji 3.x.

Analogicznie plik *3to2*, obecnie dostępny w zewnętrznej domenie, potrafi dostosować większość programów napisanych w wersji 3.x do środowiska 2.x. W zależności od celów i warunków programy *2to3* i *3to2* mogą okazać się bardzo przydatne, jeżeli trzeba utrzymywać programy napisane w obu wersjach Pythona. Więcej szczegółów na ten temat wraz z dodatkowymi narzędziami i technikami jest dostępnych w internecie.

Stosując techniki opisane w tej książce (np. importowanie funkcjonalności 3.x z *_future_*, unikanie stosowania narzędzi charakterystycznych wyłącznie dla danej wersji itp.), można pisać kod działający w wersjach 2.x i 3.x. Wiele przedstawionych w książce przykładów działa niezależnie od użytej platformy. Są to m.in. narzędzia mierzące wydajność kodu (rozdział 21.), ładujące moduły i formatujące tekst (rozdział 25.), wyświetlające zawartość drzewa klas (rozdział 31.), większość dekoratorów (rozdziały 38. i 39.), zabawny skrypt z końca rozdziału 41. i inne. Jeżeli znane są istotne różnice między wersjami 2.x a 3.x, tworzenie uniwersalnych skryptów okazuje się prostym zadaniem.

Osoby zainteresowane tworzeniem kodu przeznaczanego dla wersji 2.x i 3.x mogą zapoznać się z biblioteką *six* dostępną na stronie <http://packages.python.org/six>, umożliwiającą wiązanie i zmianianie nazw narzędzi. Oczywiście, biblioteka ta nie uwzględnia wszystkich różnic pomiędzy obydwoma wersjami i interfejsami API bibliotek, jednak w większości przypadków umożliwia napisanie uniwersalnego kodu. Dzięki niej można tworzyć programy mniej uzależnione od wersji języka.

Zmiany opisane w piątym wydaniu: wersje 2.7, 3.2 i 3.3

W tym podrozdziale opisane są zmiany wprowadzone w wersjach 2.x i 3.x Pythona po ukazaniu się czwartego wydania książki. Konkretnie dotyczy to zmian w wersjach 2.7, 3.2 i 3.3.

Zmiany w wersji 2.7

Od strony technicznej wersja Pythona 2.7 została rozszerzona o wiele funkcjonalności, które wcześniej istniały tylko w wersji 3.x i zostały opisane w poprzednim wydaniu. Narzędzia z wersji 2.7 są przedstawione również w piątym wydaniu. Są to między innymi:

- Literał zbiorów:
`{1, 4, 2, 3, 4}`
- Wyrażenia zbiorowe i słownikowe:
`{c * 4 for c in 'spam'}, {c: c * 4 for c in 'spam'}`
- Widoki słowników zaimplementowane jako opcjonalne metody:
`dict.viewkeys(), dict.viewvalues(), dict.viewitems()`
- Separatory przecinkowe i automatycznie numerowane pola w metodzie `str.format` (począwszy od wersji 3.1):
`'{:.2f} {}'.format(1234567.891, 'spam')`
- Zagnieżdżona instrukcja `with` menedżera kontekstu (począwszy od wersji 3.1):
`with X() as x, Y() as y: ...`
- Udoskonalone wyświetlanie obiektu za pomocą metody `repr` (skopiowane z wersji 3.1 — patrz dalej).

Aby dowiedzieć się, w którym miejscu książki zostały opisane powyższe funkcjonalności, należy skorzystać z przedstawionej w dalszej części niniejszego dodatku tabeli C.1 zawierającej listę zmian wprowadzonych w wersji 3.0 oraz z podrozdziału poświęconego wersji 3.1. Funkcjonalności te są dostępne w wersji 3.x, ale zostały zmienione w celu dostosowania ich do wersji 2.7.

Z organizacyjnego punktu widzenia wersja 2.7 będzie ostatnią z głównej linii 2.x. Zostanie jednak objęta długim okresem utrzymania, aby można ją było stosować w środowiskach produkcyjnych. Po upływie okresu utrzymania wersji 2.7 należy przenieść kody do wersji 3.x.

Zważywszy, że wersja 2.x jest wciąż szeroko stosowana, nie można przewidzieć, czy przyjęta polityka rozwoju Pythona przetrwa próbę czasu. Więcej informacji na ten temat jest zawartych w przedmowie. Na przykład zoptymalizowana implementacja PyPy jest na razie dostępna tylko w wersji 2.x. Mówiąc językiem Monty Pythona: „jeszcze nie umarłem”, tj. warto śledzić rozwój wersji 2.x.

Zmiany w wersji 3.3

W wersji 3.3 zostało wprowadzonych zaskakującą wiele zmian. Niektóre z nich nie są w całości kompatybilne z kodami napisanymi we wcześniejszych wersjach 3.x. Między innymi nowy program uruchomieniowy dla systemu Windows, instalowany wraz z wersją 3.3, z dużym prawdopodobieństwem może utrudniać uruchamianie skryptów w tym systemie.

Poniżej wymienione są najważniejsze zmiany wprowadzone w wersji 3.3 wraz z odniesieniami do rozdziałów, w których zostały opisane:

- Zmniejszone *zapotrzebowanie na pamięć* (dotyczy to bardziej wersji 2.x) dzięki wprowadzeniu nowego schematu przechowywania ciągów znaków oraz współdzieleniu nazw atrybutów w systemie słowników (rozdział 37. i 32.).
- Nowy *model przestrzeni nazw*, w którym można tworzyć pakiety obejmujące kilka katalogów bez korzystania z pliku `__init__.py` (rozdział 24.).
- Nowa składnia `yield from` delegowania podgeneratorów (rozdział 20.).
- Nowa składnia `raise ... from None` blokowania kontekstu wyjątków (rozdział 34.).
- Nowa składnia literałów Unicode ułatwiająca migrację kodu z wersji 2.x. Teraz w wersji 3.3 literał `u'xxxx'` jest traktowany tak samo jak zwykły ciąg `'xxxx'`. Analogicznie literał `b'xxxx'` z wersji 3.x jest w wersji 2.x traktowany jak ciąg `'xxxx'` (rozdziały 4., 7. i 37.).

- Zmieniona *hierarchia wyjątków* operacji systemowych i operacji wejścia/wyjścia, zawierająca bardziej ogólne klasy nadrzędne, jak również nowe podklasy najczęściej występujących błędów. Dzięki temu nie trzeba sprawdzać atrybutów obiektu wyjątku (rozdział 35.).
- Uniwersalny, przeglądarkowy interfejs narzędzia *dokumentacyjnego PyDoc* uruchamiany za pomocą polecenia `pydoc -b`. Zastąpił on wcześniejszy samodzielny interfejs graficzny otwierany przyciskiem *Start* w systemie Windows lub poleceniem `pydoc -g` (rozdział 15.).
- Zmiany w niektórych odwiecznych *standardowych* modułach, m.in. `ftplib`, `time`, `email` i częściowo `distutils`. W tej książce zostały opisane nowe uniwersalne odwołania do modułu `time` w wersji 3.x (rozdziały 21. i 39.).
- Bardziej ujednolicona i czytelna implementacja funkcji `__import__` w pakiecie `importlib` (rozdziały 22. i 25.).
- Nowa funkcjonalność programu instalacyjnego dla systemu Windows dopisująca do zmiennej `PATH` katalog zawierający wersję 3.3 Pythona, dzięki czemu wprowadzanie poleceń jest łatwiejsze (dodatki A i B).
- Nowy *program uruchomieniowy* `py` w systemie Windows przetwarzający uniksowe wiersze `#!`, umożliwiający wybieranie wersji 2.x lub 3.x Pythona indywidualnie dla każdego skryptu i polecenia (dodatek B).

Zmiany w wersji 3.2

Python 3.2 jest kontynuacją linii 3.x. Wersja ta została opracowana w trakcie moratorium na zmiany w rdzeniu języka, więc mało jest w niej istotnych nowości. Poniżej krótko wymienione są niektóre z nich wraz z odniesieniami do rozdziałów, w których zostały opisane:

- Zmiana w modelu przechowywania plików z kodem bajtowym: `__pycache__` (rozdziały 2 i 22.).
- Likwidacja autokodowania ciągów znaków w module `struct` (rozdziały 9. i 37.).
- Lepsza obsługa rozdzielania danych typu `str/bytes` (poza zakresem tej książki).
- Przeniesienie metody `cgi.escape` do wersji 3.2+ (poza zakresem tej książki).
- Zmiana implementacji wątków (poza zakresem tej książki).

Zmiany opisane w czwartym wydaniu: wersje 2.6, 3.0 i 3.1

W czwartym wydaniu zostały opisane wersje Pythona 3.0 i 2.6, jak również kilka większych zmian wprowadzonych w wersji 3.1. Zmiany w wersjach 3.0 i 3.1 dotyczą wszystkich następnych wersji, z 3.3 opisaną w piątym wydaniu książki włącznie. Natomiast zmiany w wersji 2.6 zostały opisane w niniejszym wydaniu w częściach poświęconych wersji 2.7. Jak wspomniałem wcześniej, niektóre opisane tu zmiany wprowadzone w wersji 3.x zostały również uwzględnione w wersji 2.7 (np. literaly zbiorów czy wyrażenia zbiorowe i słownikowe).

Zmiany w wersji 3.1

Nie licząc niżej wymienionych zmian wprowadzonych w wersjach 3.0 i 2.6, czwarte wydanie na krótko przed oddaniem do druku zostało rozszerzone o uwagi dotyczące najważniejszych rozszerzeń w nadchodzącej wersji Pythona 3.1:

- Separatory przecinkowe i automatycznie numerowane pola w metodzie `str.format` (rozdział 7.).

- Składnia menedżera kontekstu obejmująca kilka instrukcji `with` (rozdział 34.).
- Nowe metody obiektów liczbowych (rozdział 5.).
- Zmiany w wyświetlanie liczb zmiennoprzecinkowych (opisane dopiero w piątym wydaniu w rozdziałach 4. i 5.).

Powysze tematy zostały opisane w piątym wydaniu w podanych rozdziałach. Ponieważ wersja 3.1 zawierała głównie zmiany optymalizacyjne, została wprowadzona krótko po ukazaniu się wersji 3.0, zatem dotyczy jej również czwarte wydanie książki. W praktyce wersja 3.1 całkowicie zastąpiła wersję 3.0, a zazwyczaj najlepiej jest pobierać i używać najnowszej wersji, więc stosowane w tamtym wydaniu określenie „Python 3.0” dotyczyło nie tylko zmian wprowadzonych w wersji 3.0, ale również wszystkich innych w całej linii 3.x, w tym w wersji 3.3.

Jest jeden istotny wyjątek: w czwartym wydaniu nie został opisany wprowadzony w wersji 3.1 schemat wyświetlania liczb *zmiennoprzecinkowych* za pomocą metody `repr`. Nowy algorytm wyświetla liczby w bardziej intelligentnym formacie zawierającym zazwyczaj mnóstwo (a czasami więcej) cyfr po przecinku. Zmiana ta została opisana w piątym wydaniu.

Zmiany w wersjach 3.0 i 2.6

Przyczyną opracowania czwartego wydania książki były zmiany wprowadzone w wersjach Pythona 3.0 i 2.6. Wszystkie zmiany z wersji 2.6 i część z wersji 3.0 są zawarte w wersjach 2.7 i 3.3. Wersja 2.7 została rozszerzona o kilka funkcjonalności z wersji 3.0, których nie było w wersji 2.6 (opisanych wcześniej w tym dodatku). Wersja 3.3 zawiera wszystkie funkcjonalności wersji 3.0.

Ponieważ zmian wprowadzonych w początkowej wersji 3.x było bardzo wiele, poniższa tabela zawiera jedynie krótkie wzmianki o nich oraz odniesienia do rozdziałów, w których zostały opisane bardziej szczegółowo. Tabela C.1 przedstawia pierwszy zestaw zmian wprowadzonych w wersji 3.x. Są to najważniejsze nowe funkcjonalności języka opisane w czwartym wydaniu, jak również w wymienionych niżej rozdziałach piątego wydania.

Tabela C.1. Zmiany języka Python w wersjach 2.6 oraz 3.0

Zmiana	Omówiona w rozdziale (rozdziałach)
Funkcja <code>print</code> w wersji 3.0	11.
Instrukcja <code>nonlocal x,y</code> w wersji 3.0	17.
Metoda <code>str.format</code> w wersji 2.6 oraz 3.0	7.
Typy łańcuchów znaków w wersji 3.0: <code>str</code> dla tekstu Unicode, <code>bytes</code> dla danych binarnych	7., 37.
Rozróżnienie plików tekstowych i binarnych w wersji 3.0	9., 37.
Dekoratory klas w wersji 2.6 oraz 3.0: <code>@private('age')</code>	32., 39.
Nowe iteratory w wersji 3.0: <code>range</code> , <code>map</code> i <code>zip</code>	14., 20.
Widoki słowników w wersji 3.0: <code>D.keys</code> , <code>D.values</code> , <code>D.items</code>	8., 14.
Operatory dzielenia w wersji 3.0: <code>reszta</code> , <code>/</code> oraz <code>//</code>	5.
Literał zbiorów w wersji 3.0: <code>{a, b, c}</code>	5.

Zbiory składane w wersji 3.0: {x**2 for x in seq}	4., 5., 14., 20.
Słowniki składane w wersji 3.0: {x: x**2 for x in seq}	4., 8., 14., 20.
Obsługa łańcuchów cyfr binarnych w wersji 2.6 oraz 3.0: 0b0101, bin(I)	5.
Typ liczby ułamkowej w wersji 2.6 oraz 3.0: Fraction(1, 3)	5.
Adnotacje funkcji w wersji 3.0: def f(a:99, b:str)->int	19.
Argumenty mogące być tylko słowami kluczowymi w wersji 3.0: f(a, *b, c, **d)	18., 20.
Rozszerzone rozpakowywanie sekwencji w wersji 3.0: a, *b = seq	11., 13.
Składnia importowania względnego dostępna dla pakietów w wersji 3.0: from .	24.
Menedżery kontekstu dostępne w wersji 2.6 oraz 3.0: with/as	34., 36.
Zmiany w składni wyjątków w wersji 3.0: raise, except/as, klasy nadzędne	34., 35.
Łączenie wyjątków w łańcuchy w wersji 3.0: raise e2 from e1	34.
Zmiany w słowach zarezerwowanych w wersjach 2.6 oraz 3.0	11.
Przełączenie na klasy w nowym stylu w wersji 3.0	32.
Dekoratory właściwości w wersji 2.6 oraz 3.0: @właściwość	38.
Użycie deskryptorów w wersji 2.6 oraz 3.0	32., 38.
Użycie metaklas w wersji 2.6 oraz 3.0	32., 40.
Obsługa klas o abstrakcyjnej podstawie w wersji 2.6 oraz 3.0	29.

Niektóre elementy języka usunięte w Pythonie 3.0

Poza dodatkowymi opcjami w wersji 3.0 usunięto także sporą liczbę narzędzi w celu oczyszczenia jego projektu. W tabeli C.2 podsumowano zmiany, które wpłynęły na niniejsze wydanie książki, omówione w poszczególnych jej rozdziałach. Wiele z elementów usuniętych i wymienionych w tabeli C.2 ma bezpośrednie zastępniki; część z nich dostępna jest również w Pythonie 2.6 i 2.7 w celu obsłużenia przyszłej migracji do wersji 3.x.

Tabela C.2. Elementy usunięte z Pythona 3.0, które wpłynęły na niniejszą książkę

Usunięcie	Zastępnik	Omówione w rozdziale (rozdziałach)
reload(M)	imp.reload(M) (lub exec)	3., 23.

<code>apply(f, ps, ks)</code>	<code>f(*ps, **ks)</code>	18.
<code>`X`</code>	<code>repr(X)</code>	5.
<code>X <= Y</code>	<code>X != Y</code>	5.
<code>long</code>	<code>int</code>	5.
<code>9999L</code>	<code>9999</code>	5.
<code>D.has_key(K)</code>	<code>K in D (lub D.get(key) != None)</code>	8.
<code>raw_input</code>	<code>input</code>	3., 10.
<code>old input</code>	<code>eval(input())</code>	3.
<code>xrange</code>	<code>range</code>	13, 14.
<code>file</code>	<code>open (oraz klasy modułu io)</code>	9.
<code>X.next</code>	<code>X.__next__, wywoływanie za pomocą next(X)</code>	14., 20., 30.
<code>X.__getslice__</code>	<code>X.__getitem__ po przekazaniu obiektu wycinka</code>	7., 30.
<code>X.__setslice__</code>	<code>X.__setitem__ po przekazaniu obiektu wycinka</code>	7., 30.
<code>reduce</code>	<code>functools.reduce (lub kod pętli)</code>	14., 19.
<code>execfile(nazwa_pliku)</code>	<code>exec(open(nazwa_pliku).read())</code>	3.
<code>exec open(nazwa_pliku)</code>	<code>exec(open(nazwa_pliku).read())</code>	3.
<code>0777</code>	<code>0o777</code>	5.
<code>print x, y</code>	<code>print(x, y)</code>	11.
<code>print >> F, x, y</code>	<code>print(x, y, file=F)</code>	11.
<code>print x, y,</code>	<code>print(x, y, end=' ')</code>	11.
<code>u'ccc' (wstecz w wersji 3.3)</code>	<code>'ccc'</code>	4., 7., 36.
<code>'bbb' dla łańcuchów bajtowych</code>	<code>b'bbb'</code>	4., 7., 9., 36.
<code>raise E, V</code>	<code>raise E(V)</code>	33., 34., 35.
<code>except E, X:</code>	<code>except E as X:</code>	33., 34., 35.
<code>def f((a, b)):</code>	<code>def f(x): (a, b) = x</code>	11., 18., 20.
<code>file.xreadlines</code>	<code>for line in file: (lub X=iter(file))</code>	13., 14.
<code>D.keys() itd. jako listy</code>	<code>list(D.keys()) (widoki słowników)</code>	8., 14.

<code>map()</code> , <code>range()</code> itd. jako listy	<code>list(map()), list(range())</code> (funkcje wbudowane)	14.
<code>map(None, ...)</code>	<code>zip</code> (lub ręczny kod dopełniający wyniki)	13., 20.
<code>X=D.keys(); X.sort()</code>	<code>sorted(D)</code> (lub <code>list(D.keys())</code>)	4., 8., 14.
<code>cmp(x, y)</code>	<code>(x > y) - (x < y)</code>	30.
<code>X.__cmp__(y)</code>	<code>__lt__, __gt__, __eq__</code> itd.	30
<code>X.__nonzero__</code>	<code>X.__bool__</code>	30.
<code>X.__hex__, X.__oct__</code>	<code>X.__index__</code>	30.
Sortujące funkcje porównujące	Należy użyć <code>key=transform</code> lub <code>reverse=True</code>	8.
Operacje <code><</code> , <code>></code> , <code><=</code> , <code>>=</code> dla słowników	Porównanie <code>sorted(D.items())</code> (lub kod pętli)	8., 9.
<code>types.ListType</code>	<code>list</code> (<code>types</code> przeznaczony jest jedynie dla nazw niewbudowanych)	9.
<code>__metaclass__ = M</code>	<code>class C(metaclass=M):</code>	29., 32., 40.
<code>__builtin__</code>	<code>builtins</code> (zmiana nazwy)	17.
<code>Tkinter</code>	<code>tkinter</code> (zmiana nazwy)	18., 19., 25., 30., 31.
<code>sys.exc_type, exc_value</code>	<code>sys.exc_info()[0], [1]</code>	35., 36.
<code>function.func_code</code>	<code>function.__code__</code>	19., 39.
<code>__getattr__</code> wykonywane przez obiekty wbudowane	Ponowne zdefiniowane metod <code>__X__</code> w klasach opakowujących	31., 38., 39.
Opcje wiersza poleceń <code>-t</code> , <code>-tt</code>	Niespójne użycie tabulatorów i spacji zawsze jest błędem	10., 12.
<code>from ... * wewnątrz funkcji</code>	Może się pojawiać jedynie na najwyższym poziomie pliku	23.
<code>import mod w tym samym pakiecie</code>	<code>from . import mod</code> , forma względem pakietu	24.
<code>class MyException:</code>	<code>class MyException(Exception):</code>	35.
Moduł <code>exceptions</code>	Zakres wbudowany, dokumentacja biblioteki	35.
Moduły <code>thread</code> i <code>Queue</code>	<code>_thread, queue</code> (zmiany nazw)	17.
Moduł <code>anydbm</code>	<code>dbm</code> (zmiana nazwy)	28.
Moduł <code>cPickle</code>	<code>_pickle</code> (zmiana nazwy,	9.

	wykorzystywany automatycznie)	
os.popen2/3/4	subprocess.Popen (zachowano os.popen)	14.
Wyjątki oparte na łańcuchach znaków	Wyjątki oparte na klasach (wymagane także w wersji 2.6)	33., 34., 35.
Funkcje modułu łańcuchów znaków	Metody obiektu łańcuchów znaków	7.
Metody bez wiązania	Funkcje (staticmethod wywoływanego za pośrednictwem instancji)	31., 32.
Porównywanie i sortowanie obiektów o typach mieszanych	Porównania mieszanych typów nieliczbowych są błędem	5., 9.

Zmiany opisane w trzecim wydaniu: wersje 2.3, 2.4 i 2.5

Trzecie wydanie książki zostało uaktualnione do wersji Pythona zanumerowanej jako 2.5; wprowadzono również inne zmiany, które pojawiły się po wydaniu drugiej edycji książki z 2003 roku. Drugie wydanie książki bazowało z kolei na wersji 2.2; na końcu projektu dodano również niektóre elementy z Pythona 2.3. W odpowiednich miejscach trzeciego wydania wstawione zostały również uwagi dotyczące zbliżającej się wersji 3.0. Poniżej znajduje się lista najważniejszych zagadnień, które zostały wprowadzone bądź uaktualnione w tej wersji książki (numery rozdziałów zostały uaktualnione i mają zastosowanie do wydania piątego):

- wyrażenie warunkowe `B if A else C` (rozdział 12. oraz 19.),
- menedżery kontekstu — `with/as` (rozdział 34.),
- ujednolicenie `try/except/finally` (rozdział 34.),
- składnia importowania względnego (rozdział 24.),
- wyrażenia generatora (rozdział 20.),
- nowe możliwości funkcji generatora (rozdział 20.),
- dekoratory funkcji (rozdziały 32. oraz 39.),
- typ obiektu zbioru (rozdział 5.),
- nowe funkcje wbudowane — `sorted`, `sum`, `any`, `all`, `enumerate` (rozdziały 13. oraz 14.),
- typ obiektu liczby dziesiętnej o stałej precyzyji (rozdział 5.),
- pliki, listy składane oraz iteratory (rozdziały 14. oraz 20.),
- nowe narzędzia programistyczne — między innymi `Eclipse`, `distutils`, `unittest` oraz `doctest`, ulepszenia `IDLE`, `Shedskin` (rozdział 2. oraz 36.).

Mniejsze zmiany w języku Python (na przykład rozszerzone użycie `True` oraz `False`, nowa funkcja `sys.exc_info` służąca do pobierania szczegółowych informacji o wyjątkach, zniknięcie wyjątków opartych na łańcuchach znaków, metod działających na łańcuchach znaków, wbudowanych funkcji `apply` oraz `reduce`) omawiane były w całej książce. Niektóre zagadnienia, które były nowością w wydaniu drugim, w wydaniu trzecim zostały omówione bardziej szczegółowo — w tym wycinki o trzech granicach oraz składnia wywołania o dowolnej liczbie argumentów, która zastąpiła funkcję `apply`.

Wcześniejsze i późniejsze zmiany w Pythonie

W poprzednich wydaniach książki również zostały opisane zmiany wprowadzone w Pythonie. Pierwsze dwa wydania z lat 1999 i 2003 obejmowały wersje 2.0 i 2.2, a pierwowzór, tj. pierwsze wydanie książki *Programming Python* z 1996, z którego wywodzą się trzy następne książki, był poświęcony wersji 1.3. Wydania te pominąłem w tym dodatku, ponieważ dotyczą zamierzczej przeszłości (przynajmniej w dziedzinie programowania).

Zainteresowani czytelnicy znajdą więcej szczegółów w wydaniach pierwszym i drugim. Nie sposób wprawdzie przewidzieć przyszłości, ale zważywszy, że najważniejsze opisane w tej książce funkcjonalności przetrwały próbę czasu, można sądzić, że będą one obecne również w przyszłych wersjach Pythona.

Dodatek D Rozwiązań ćwiczeń podsumowujących poszczególne części książki

Część I Wprowadzenie

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części pierwszej” na końcu rozdziału 3.

1. *Interakcja.* Zakładając, że Python jest poprawnie skonfigurowany, interakcja powinna wyglądać mniej więcej tak, jak poniżej — sesję interaktywną można uruchomić w dowolny sposób, na przykład w IDLE czy z powłoki systemowej:

```
% python  
...wiersze z informacjami o prawach autorskich...  
>>> "Witaj, świecie!"  
'Witaj, świecie!'  
>>> # By wyjść z sesji interaktywnej, należy  
użyć Ctrl+D lub Ctrl+Z bądź zamknąć okno.
```

2. *Programy.* Plik z kodem (czyli moduł) o nazwie *module1.py* wraz z działaniami wykonywanymi w powłoce systemowej powinien wyglądać następująco:

```
print('Witaj, module!')  
% python module1.py  
Witaj, module!
```

Można uruchomić ten plik na inne sposoby — na przykład klikając jego ikonę czy korzystając z opcji *Run Module* z menu *Run* programu IDLE.

3. *Moduły.* Poniższy listing kodu z sesji interaktywnej ilustruje wykonywanie kodu modułu za pomocą importowania:

```
% python  
>>> import module1  
Witaj, module!  
>>>
```

Należy pamiętać, że by wykonać moduł ponownie bez zatrzymywania i ponownego uruchamiania interpretera, należy moduł przeładować. Pytanie z przeniesieniem pliku do innego katalogu i ponownym go zaimportowaniem jest podchwytliwe — jeśli Python wygenerował plik *module1.pyc* w oryginalnym katalogu, wykorzysta go przy

imporcie modułu, nawet jeśli plik z kodem źródłowym (.py) zostanie przeniesiony do katalogu niebędącego w ścieżce wyszukiwania Pythona. Plik .pyc jest zapisywany automatycznie, jeśli Python ma dostęp do katalogu pliku źródłowego, i zawiera skompilowany kod bajtowy modułu. Więcej informacji na temat modułów znajduje się w rozdziale 3.

4. *Skrypty*. Zakładając, że nasza platforma obsługuje sztuczkę z #!, rozwiążanie powinno wyglądać jak poniżej (choć nasz wiersz ze znakami #! może zawierać inną ścieżkę). Należy pamiętać, że wiersz z tymi znakami ma znaczenie wtedy, gdy skrypt jest uruchamiany za pomocą programu uruchomieniowego dostarczanego i instalowanego z wersją 3.3 Pythona. Taki wiersz określa wersję interpretera i jego domyślne ustawienia. Szczegółowe informacje na ten temat i przykłady znajdują się w dodatku B.

```
#!/usr/local/bin/python                                (lub #!/usr/bin/env  
python)  
  
print('Witaj, module!')  
  
% chmod +x module1.py  
  
% module1.py  
  
Witaj, module!
```

5. *Błędy*. Poniższa sesja interaktywna (wykonana w Pythonie 3.x) demonstruje rodzaj komunikatów o błędzie, jakie możnatrzymać w trakcie wykonywania ćwiczenia. Tak naprawdę wywołujemy wyjątki Pythona. Domyślne zachowanie w momencie wystąpienia błędu kończy wykonywanie programu w Pythonie i wyświetla komunikat o błędzie wraz ze śladem stosu. Stos wywołań pokazuje, w którym miejscu programu byliśmy w momencie wystąpienia wyjątku (jeśli wywołania funkcji są aktywne, kiedy występuje błąd, część „Traceback” wyświetla wszystkie aktywne poziomy wywołań). W rozdziale 10. i w siódmej części książki pokażemy, że wyjątki można przechwytywać za pomocą instrukcji try, a później przetwarzać je w dowolny sposób. Jak się zresztą okaże, Python zawiera debugger kodu źródłowego, który świetnie sprawdza się przy specjalnych wymaganiach w zakresie wykrywania błędów. Jak na razie warto zauważyc, że Python w momencie wystąpienia błędów podaje znaczące komunikaty (a nie po cichu kończy działanie programu).

```
% python  
=>> 2 ** 500  
327339060789614187001318969682759915221664204604306478948329136809  
613379640467455488327009232590415715088668412756007100921725654588  
5393053328527589376  
  
>>>  
>>> 1 / 0  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: int division or modulo by zero  
  
>>>  
>>> spam  
Traceback (most recent call last):
```

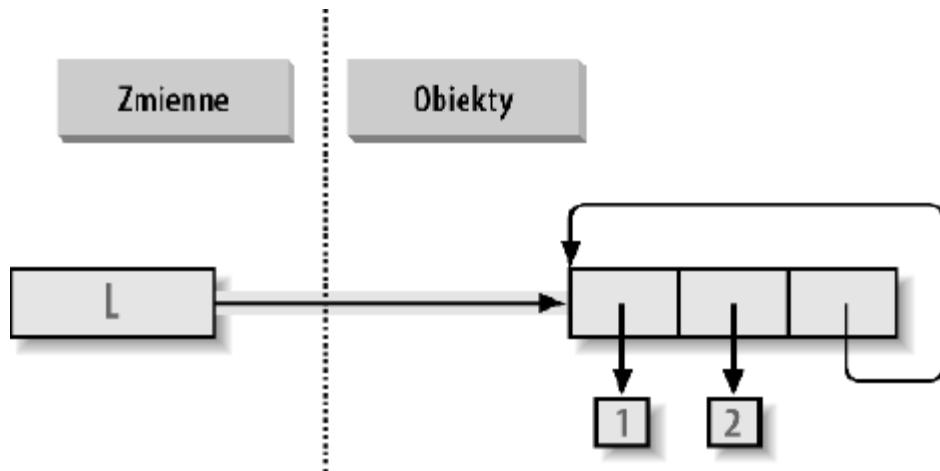
```
File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

6. Przerwania i cykle. Kiedy wpiszemy poniższy kod:

```
L = [1, 2]
L.append(L)
```

tworzymy w Pythonie cykliczną strukturę danych. W wersjach Pythona starszych od 1.5.1 cykle w obiektach nie były wykrywane, przez co Python wyświetlał niekończący się strumień `[1, 2, [1, 2, [1, 2, [1, 2, i tak dalej — aż do wywołania kombinacji klawiszy przerywającej działanie programu (co z technicznego punktu widzenia powoduje wystąpienie wyjątku związanego z przerwaniem i wyświetlenie standardowego komunikatu). Od wersji 1.5.1 Python wykrywa cykle i wyświetla zamiast poprzedniego zapisu znaki [[...]], by nas poinformować o wykryciu pętli w strukturze obiektu i uniknięciu utknięcia związanego z nieskończonymi próbami wyświetlania.`

Powód wystąpienia cyklu jest dość subtelny i wymaga informacji, które będziemy omawiać w drugiej części książki, dlatego na razie jedynie o tym wspomnimy. Mówiąc w skrócie, przypisanie w Pythonie zawsze generuje *referencje* do obiektów, a nie ich kopie. Obiekty możemy sobie wyobrazić jako fragmenty pamięci, natomiast referencje — jako niejawne wskaźniki, którymi podążamy. Kiedy wykonujemy pierwsze przypisanie, zmienna `L` staje się nazwaną referencją do obiektu listy dwuelementowej — wskaźnikiem do miejsca w pamięci. Listy Pythona to tak naprawdę tablice referencji do obiektów z metodą `append`, która zmienia tablicę w miejscu, dołączając do niej (na końcu) inną referencję do obiektu. Tutaj metoda `append` dodaje po `L` referencję do `L`, co prowadzi do cyklu widocznego na rysunku D.1: na końcu listy znajduje się wskaźnik, który kieruje z powrotem do początku listy.



Rysunek D.1. Obiekt cykliczny utworzony poprzez dodanie listy do samej siebie. Domyslnie Python dodaje referencje do oryginalnej listy, a nie kopię tej listy

Poza specjalnym sposobem wyświetlania obiekty cykliczne muszą również być w specjalny sposób obsługiwane przez mechanizm czyszczenia pamięci Pythona (o czym dowiemy się w rozdziale 6.) — inaczej zajmowane przez nie miejsce nie zostanie zwolnione, kiedy przestaną one być w użyciu. Mimo że w praktyce jest to stosunkowo rzadkie, w niektórych programach przechodzących obiekty lub struktury

wykrywanie cykli może być konieczne, by zapobiec zapętleniu. Choć trudno w to uwierzyć, cykliczne struktury danych mogą czasami być użyteczne, pomimo że wyświetlane są w specjalny sposób.

Część II Typy i operacje

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części drugiej” na końcu rozdziału 9.

1. *Podstawy.* Poniżej widać rezultaty, jakie można otrzymać, wraz z kilkoma komentarzami dotyczącymi ich znaczenia. Ponownie warto zwrócić uwagę, że w kilku przypadkach wykorzystano znak ; w celu zmieszczenia większej liczby instrukcji w jednym wierszu (znak ; jest separatorem instrukcji), natomiast przecinki tworzą krotki wyświetlane w nawiasach. Należy również pamiętać, że wynik operacji dzielenia na górze kodu będzie różny dla Pythona 2.x oraz 3.x (więcej informacji na ten temat można znaleźć w rozdziale 5.), natomiast opakowanie wywołań metod słownika w list jest niezbędne do wyświetlenia wyników w wersji 3.x, jednak w 2.x już nie (więcej o tym w rozdziale 8.).

```
# Liczby
>>> 2 ** 16                                # 2 do potęgi 16
65536
>>> 2 / 5, 2 / 5.0                         # Liczba całkowita – odcięcie w
Pythonie 2.x, jednak w wersji 3.x już nie
(0.4000000000000002, 0.4000000000000002)
# Łańcuchy znaków
>>> "mielonka" + "jajka"                  # Konkatenacja
'mielonkajajka'
>>> S = "szynka"
>>> "jajka " + S
'jajka szynka'
>>> S * 5                                    # Powtórzenie
'szynkaszynkaszynkaszynkaszynka'
>>> S[:0]                                    # Pusty wycinek z przodu – [0:0]
''                                         # Pusty wycinek tego samego typu
co pocięty obiekt
>>> "zielone %s i %s" % ("jajka", S)      # Formatowanie
'zielone jajka i szynka'
>>> 'zielone {0} i {1}'.format('jajka', S)
'zielone jajka i szynka'
# Krotki
```

```

>>> ('x',)[0]                                # Indeksowanie krotki
jednoelementowej
'x'

>>> ('x', 'y')[1]                            # Indeksowanie krotki
dwuelementowej
'y'

# Listy
>>> L = [1,2,3] + [4,5,6]      # Operacje na listach
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1,2,3]+[4,5,6])[2:4]
[3, 4]

>>> [L[2], L[3]]                          # Pobranie elementów o wartości
przesunięcia; przechowanie w liście
[3, 4]

>>> L.reverse(); L                         # Metoda: odwracanie listy w
miejscu
[6, 5, 4, 3, 2, 1]

>>> L.sort(); L                           # Metoda: sortowanie listy w
miejscu
[1, 2, 3, 4, 5, 6]

>>> L.index(4)                           # Metoda: wartość przesunięcia
pierwszego elementu 4 (wyszukiwanie)
3

# Słowniki
>>> {'a':1, 'b':2}['b']                  # Indeksowanie słownika po kluczu
2

>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                           # Utworzenie nowego wpisu
>>> D['x'] + D['w']
1

>>> D[(1,2,3)] = 4                      # Krotka użyta jako klucz (jest
niezmienna)
>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}

>>> list(D.keys()), list(D.values()), (1,2,3) in D    # Metody,
sprawdzenie kluczy
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], True)

```

```
# Puste obiekty
>>> [[], "", (), {}, None]      # Dużo niczego: puste obiekty
([[], ['', [], (), {}, None])
```

2. *Indeksowanie i wycinki.* Indeksowanie poza granicami listy (na przykład `L[4]`) powoduje wystąpienie błędu. Python zawsze sprawdza, czy wszystkie wartości przesunięcia mieszczą się w granicach sekwencji.

Z drugiej strony, wycinek wykraczający poza granice listy (jak `L[-1000:100]`) działa, ponieważ Python skaluje wycinki tego rodzaju, tak by zawsze pasowały (w razie konieczności granice ustawiane są na zero i długość sekwencji).

Ekstrakcja sekwencji w odwrotnej kolejności, z niższą granicą większą od wyższej (na przykład `L[3:1]`), nie działa. Z powrotem otrzymamy pusty wycinek (`[]`), ponieważ Python skaluje granice wycinka w celu upewnienia się, że niższa granica jest zawsze mniejsza od lub równa wyższej granicy (`L[3:1]` zawsze skalowane jest do `L[3:3]`, pustego miejsca wstawiania w pozycji przesunięcia o wartości 3). Wycinki Pythona zawsze dokonywane są od lewej strony do prawej, nawet jeśli skorzystamy z ujemnych indeksów (najpierw są one przekształcane na indeksy dodatnie poprzez dodanie długości sekwencji). Warto zauważyć, że zachowanie to modyfikują nieco wycinki z trzema argumentami wprowadzone w Pythonie 2.3 — na przykład `L[3:1:-1]` powoduje ekstrakcję elementów od prawej strony do lewej:

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. *Indeksowanie, wycinki i del.* Nasza interakcja z interpreterem Pythona powinna wyglądać podobnie do poniższego listingu. Warto zauważyć, że przypisanie pustej listy do wartości przesunięcia powoduje umieszczenie tam pustej listy, natomiast przypisanie pustej listy do wycinka usuwa ten wycinek. Przypisanie do wycinka oczekuje innej sekwencji — w przeciwnym razie otrzymuje się błąd typu. Wstawia ono elementy *do środka* przypisanej sekwencji, a nie do samej sekwencji.

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
```

```

[1, 2, [], 4]
>>> L[2:3] = []
>>> L
[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
>>> L[1:2] = 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation

```

4. *Przypisywanie krotek.* Wartości X oraz Y zostają zamienione. Kiedy krotki pojawiają się po lewej i prawej stronie symbolu przypisania (=), Python przypisuje obiekty znajdujące się po prawej stronie do obiektów docelowych z lewej strony, zgodnie z ich pozycjami. Najłatwiej będzie to chyba zrozumieć, uwzględniając fakt, że obiekt docelowy z lewej strony nie jest prawdziwą krotką, nawet jeśli tak wygląda — tak naprawdę to zbiór niezależnych celów przypisania. Elementy z prawej strony są krotką, która przy przypisaniu zostaje rozpakowana (krotka udostępnia tymczasowe przypisanie potrzebne do uzyskania efektu zamiany).

```

>>> X = 'mielonka'
>>> Y = 'jajka'
>>> X, Y = Y, X
>>> X
'jajka'
>>> Y
'mielonka'

```

5. *Klucze słowników.* Kluczem słownika może być dowolny obiekt niezmienny — w tym liczby całkowite, krotki czy łańcuchy znaków. To naprawdę jest słownik, choć niektóre z jego kluczy wyglądają jak wartości przesunięcia będące liczbami całkowitymi. Zadziałają również klucze o mieszanych typach.

```

>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D

```

```
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Indeksowanie słowników.* Próba zindeksowania nieistniejącego klucza (`D['d']`) kończy się błędem. Przypisanie do nieistniejącego klucza (`D['d']='mielonka'`) tworzy nowy wpis do słownika. Indeksowanie listy za pomocą nieistniejącej wartości przesunięcia również zwraca błąd, podobnie jak próba przypisania do takiej wartości. Nazwy zmiennych działają jak klucze słowników — przy referencji muszą już być przypisane i są tworzone przy pierwszym przypisaniu. Nazwy zmiennych można tak naprawdę przetwarzarć jako klucze słowników, jeśli mamy na to ochotę (są one widoczne w przestrzeni nazw modułu lub słownikach ramki stosu).

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D['a']
1
>>> D['d']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: d
>>> D['d'] = 4
>>> D
{'b': 2, 'd': 4, 'a': 1, 'c': 3}
>>>
>>> L = [0, 1]
>>> L[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

7. *Operacje uniwersalne.* Odpowiedzi na pytania:

- Operator `+` nie działa na różnych (mieszanych) typach obiektów (na przykład łańcuchu znaków i liście czy liście i krotce).
- Operator `+` nie działa dla słowników, ponieważ nie są one sekwencjami.
- Metoda `append` działa tylko dla list, a nie łańcuchów znaków; metoda `keys` działa tylko dla słowników. Metoda `append` zakłada, że jej obiekt docelowy jest zmienny, ponieważ jest rozszerzeniem w miejscu — łańcuchy znaków są niezmienne.
- Wycinki i konkatenacja zawsze zwracają nowy obiekt tego samego typu co obiekt przetwarzany.

```
>>> "x" + 1
Traceback (innermost last):
```

```

        File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
>>>
>>> {} + {}
Traceback (innermost last):
        File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
        File "<stdin>", line 1, in ?
AttributeError: 'str' object has no attribute 'append'
>>>
>>> list({}.keys())                      # Wywołanie list()
niezbędne jest w Pythonie 3.x, w 2.x nie
[]
>>> [].keys( )
Traceback (innermost last):
        File "<stdin>", line 1, in ?
AttributeError: 'list' object has no attribute 'keys'
>>>
>>> [][:]
[]
>>> ""[:]
''
```

8. *Indeksowanie łańcuchów znaków.* To nieco podchwytliwe pytanie — ponieważ łańcuchy znaków są kolekcjami łańcuchów jednoznakowych, przy ich indeksowaniu otrzymujemy z powrotem łańcuch znaków, który znowu można zindeksować. S[0][0][0][0] po prostu ciągle indeksuje pierwszy znak. Nie działa to dla list (listy mogą przechowywać dowolne obiekty), o ile nie zawierają one łańcuchów znaków.

```

>>> S = "mielonka"
>>> S[0][0][0][0][0]
'm'
>>> L = ['m', 'i']
>>> L[0][0][0]
```

'm'

9. *Typy niezmienne.* Zadziała dowolne z poniższych rozwiązań. Nie zadziała przypisanie do indeksu, ponieważ łańcuchy znaków są niezmienne.

```
>>> S = "jajo"  
>>> S[0:3] + 'a'  
>>> S  
'jaja'  
>>> S[0] + S[1] + S[2] + 'a'  
>>> S  
'jaja'
```

(Polecam również zająrzyć do omówienia typu łańcuchów znaków `bytarray` z Pythona 3.x w rozdziale 37. — jest on zmienną sekwencją małych liczb całkowitych, przetwarzaną właściwie w ten sam sposób co zwykłe łańcuchy znaków).

10. *Zagnieżdżanie.* Poniżej przykład:

```
>>> me = {'name':('Robert', 'F', 'Zielony'), 'age':'?',  
'job':'inżynier'}  
>>> me['job']  
'inżynier'  
>>> me['name'][2]  
'Zielony'
```

11. *Pliki.* Poniżej zaprezentowano sposób na utworzenie i wczytanie pliku tekstowego w Pythonie (`ls` to polecenie z Uniksa; w systemie Windows należy skorzystać z `dir`).

```
# Plik maker.py  
  
file = open('myfile.txt', 'w')  
  
file.write('Witaj, wspaniały świecie!\n')      # Lub: open( ).write()  
)  
  
file.close()                                     # close nie zawsze  
jest potrzebne  
  
# Plik reader.py  
  
file = open('myfile.txt')                         # 'r' to domyślny  
tryb otwierania pliku  
  
print(file.read())                                # Lub: print(open()  
)  
  
% python maker.py  
% python reader.py  
  
Witaj, wspaniały świecie!  
% ls -l myfile.txt  
-rwxrwxrwa    1 0          0          19 Apr 13 16:33 myfile.txt
```

Część III Instrukcja i składnia

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części trzeciej” na końcu rozdziału 15.

1. *Zapisywanie w kodzie podstawowych pętli.* W miarę wykonywania tego ćwiczenia powinniśmy otrzymać kod wyglądający w następujący sposób:

```
>>> S = 'mielonka'  
>>> for c in S:  
...     print(ord(c))  
  
...  
115  
112  
97  
109  
>>> x = 0  
>>> for c in S: x += ord(c)          # Lub: x = x + ord(c)  
  
...  
>>> x  
433  
>>> x = []  
>>> for c in S: x.append(ord(c))  
  
...  
>>> x  
[115, 112, 97, 109]  
>>> list(map(ord, S))             # Wywołanie list() wymagane  
jest w Pythonie 3.x, jednak nie w 2.x  
[115, 112, 97, 109]  
>>> [ord(c) for c in S]           # Mapa i wyrażenie listowe  
automatyzuje tworzenie listy  
[115, 112, 97, 109]
```

2. *Znaki ukośników lewych.* Przykład wyświetla pięćdziesiąt razy znak sygnału dźwiękowego (\a). Zakładając, że nasz komputer potrafi sobie z tym poradzić, i kiedy wykonujemy kod poza IDLE, możemy otrzymać serię pisków (lub jeden długi dźwięk, jeśli nasz komputer jest wystarczająco szybki). Ostrzegałem!
3. *Sortowanie słowników.* Poniżej znajduje się jeden ze sposobów wykonania ćwiczenia (jeśli wydaje się on nie mieć sensu, należy spojrzeć do rozdziału 8. lub rozdziału 14.).

Należy pamiętać, że naprawdę musimy rozbić wywołania metod `keys` i `sort` w poniższy sposób, gdyż metoda `sort` zwraca `None`. W Pythonie 2.2 i późniejszych wersjach można wykonać iterację bezpośrednio po kluczach słownika bez wywoływania metody `keys` (za pomocą `for key in D:`), jednak lista kluczy nie zostanie posortowana tak, jak w kodzie niżej. W nowszych wersjach Pythona ten sam efekt można uzyskać za pomocą wbudowanej metody `sorted`.

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = list(D.keys())           # Wywołanie list() wymagane
jest w Pythonie 3.x, jednak nie w 2.x
>>> keys.sort( )
>>> for key in keys:
...     print(key, '=>', D[key])
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7
>>> for key in sorted(D):          # Lepsze rozwiązanie, w
nowszych wersjach Pythona
...     print(key, '=>', D[key])
```

4. *Alternatywne logiki programu.* Poniżej znajdują się różne wersje kodu będące rozwiązaniem. W przypadku kroku e należy przypisać wynik działania $2^{**} X$ do zmiennej poza pętlami z kroków a oraz b i użyć tej zmiennej wewnętrz pętli. Każda osoba może mieć nieco inne wyniki. To ćwiczenie zostało zaprojektowane przede wszystkim jako umożliwiające zabawę z alternatywnymi wersjami kodu, dlatego punkty można otrzymać za każde rozsądne rozwiązanie.

```
# a
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('pod indeksem', i)
```

```

        break
    i = i+1
else:
    print(X, 'nie odnaleziono')

# b
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'odnaleziono pod indeksem', L.index(p))
        break
else:
    print(X, 'nie odnaleziono')

# c
L = [1, 2, 4, 8, 16, 32, 64]
X = 5
if (2 ** X) in L:
    print((2 ** X), 'odnaleziono pod indeksem', L.index(2 ** X))
else:
    print(X, 'nie odnaleziono')

# d
X = 5
L = []
for i in range(7): L.append(2 ** i)
print(L)
if (2 ** X) in L:
    print((2 ** X), 'odnaleziono pod indeksem', L.index(2 ** X))
else:
    print(X, 'nie odnaleziono')

# e
X = 5
L = list(map(lambda x: 2**x, range(7)))      # Lub [2**x for x in
                                                # range(7)] list() w celu
print(L)                                         # wyświetlenia
wszystkiego w Pythonie 3.0, w 2.6 nie

```

```

if (2 ** X) in L:
    print((2 ** X), 'odnaleziono pod indeksem', L.index(2 ** X))
else:
    print(X, 'nie odnaleziono')

```

5. *Utrzymanie kodu.* Nie ma uniwersalnego rozwiązania, które można byłoby tu pokazać. Jedno z przykładowych, przeze mnie napisanych, jest zawarte w pliku *mypydoc.py* dołączonym do tej książki.

Część IV Funkcje i generatory

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części czwartej” na końcu rozdziału 21.

1. *Podstawy.* W tym ćwiczeniu nie ma nic specjalnego, jednak warto zauważyć, że użycie instrukcji `print` (i tym samym naszej funkcji) jest z technicznego punktu widzenia operacją *polimorficzną*, która wykonuje odpowiednie działanie dla każdego typu obiektu.

```

% python
>>> def func(x): print(x)
...
>>> func("mielonka")
mielonka
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'jedzenie': 'mielonka'})
{'jedzenie': 'mielonka'}

```

2. *Argumenty.* Poniżej znajduje się przykładowe rozwiązanie. Należy pamiętać, że by zobaczyć wyniki z wywołań testowych, musimy użyć instrukcji `print`, ponieważ plik nie jest tym samym co kod wpisany w sesji interaktywnej. Python nie zwraca normalnie wyników instrukcji wyrażeń w plikach.

```

def adder(x, y):
    return x + y
print(adder(2, 3))
print(adder('mielonka', 'jajka'))
print(adder(['a', 'b'], ['c', 'd']))
% python mod.py

```

5

```
mielonkajajka
['a', 'b', 'c', 'd']
```

3. *Zmienna liczba argumentów.* Dwie alternatywne wersje funkcji adder zaprezentowane zostały w pliku *adders.py*. Najtrudniejszym zadaniem jest tu znalezienie sposobu na zainicjalizowanie akumulatora na pustą wartość dowolnego przekazywanego typu. Pierwsze rozwiązanie wykorzystuje ręczne sprawdzanie typów w celu odszukania liczby całkowitej oraz pusty wycinek pierwszego argumentu (który, zgodnie z założeniem, ma być sekwencją), jeśli argument okazuje się nie być liczbą całkowitą. Drugie rozwiązanie wykorzystuje pierwszy argument do zainicjalizowania i przejrzenia elementów od drugiego w górę, podobnie do wariantów funkcji min zaprezentowanych w rozdziale 18.

Drugie rozwiązanie jest lepsze. Oba zakładają, że wszystkie argumenty są tego samego typu, i żadne nie działa na słownikach (jak widzieliśmy w drugiej części książki, operator + nie działa na typach mieszanych oraz słownikach). Moglibyśmy również dodać sprawdzanie typów i specjalny kod obsługujący słowniki, jednak jest to zadanie dodatkowe.

```
def adder1(*args):
    print('adder1', end=' ')
    if type(args[0]) == type(0):                      # Liczba całkowita?
        sum = 0                                         # Inicjalizacja do wartości zero
    else:                                              # Inaczej sekwencja
        sum = args[0][:0]                                # Wykorzystanie
        pustego wycinka arg1
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print('adder2', end=' ')
    sum = args[0]                                      # Inicjalizacja do arg1
    for next in args[1:]:
        sum += next                                     # Dodanie elementów
    return sum

for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('mielonka', 'jajka', 'tost'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))
% python adders.py
```

```

adder1 9
adder1 mielonkajakatost
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 mielonkajakatost
adder2 ['a', 'b', 'c', 'd', 'e', 'f']

```

4. *Słowa kluczowe.* Poniżej znajduje się moje rozwiązańe pierwszej i drugiej części ćwiczenia (plik *mod.py*). By wykonać iterację po argumentach będących słowami kluczowymi, należy w nagłówku funkcji użyć formy **args* i wykorzystać pętlę (na przykład *for x in args.keys(): use args[x]*) lub użyć *args.values()*, by przypominało to sumowanie pozycyjnych argumentów **args*.

```

def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly
print(adder( ))
print(adder(5))
print(adder(5, 6))
print(adder(5, 6, 7))
print(adder(ugly=7, good=6, bad=5))
% python mod.py
6
10
14
18
18
# Rozwiązańie części drugiej
def adder1(*args):          # Suma dowolnej liczby argumentów
    pozycyjnych
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot
def adder2(**args):          # Suma dowolnej liczby argumentów
    ze słowami kluczowymi
    argskeys = list(args.keys()) # W Pythonie 3.0 dodano wywołanie
    list()!
    tot = args[argskeys[0]]
    for key in argskeys[1:]:

```

```

        tot += args[key]

    return tot

def adder3(**args):           # To samo, ale przekształcenie na
    liste wartości

    args = list(args.values())   # W Pythonie 3.0 do indeksowania
    dodano wywołanie list!

    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):           # To samo, ale z ponownym użyciem
    wersji z argumentami pozycyjnymi

    return adder1(*args.values())

print(adder1(1, 2, 3),      adder1('aa', 'bb', 'cc'))
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb', c='cc'))
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb', c='cc'))
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb', c='cc'))

```

5. (oraz 6.). *Narzędzia słownika.* Poniżej znajdują się moje rozwiązania ćwiczenia 5. oraz 6. (plik *dicts.py*). Są to jedynie ćwiczenia programistyczne, gdyż w Pythonie 1.5 dodano metody słowników *D.copy()* oraz *D1.update(D2)*, które obsługują działania takie, jak kopiowanie oraz dodawanie (łączenie) słowników. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona lub w książce *Python. Leksykon kieszonkowy. Wydanie IV* wydawnictwa Helion. *X[:]* nie działa na słownikach, gdyż nie są one sekwencjami (więcej informacji w rozdziale 8.). Należy również pamiętać, że jeśli wykonamy przypisanie (*e = d*) zamiast kopiowania, wygenerujemy referencję do *współdzielonego* obiektu słownika, a modyfikacja *d* pociąga za sobą zmianę *e*.

```

def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new

```

```
% python
>>> from dicts import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}
>>> x = {1: 1}
>>> y = {2: 2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}
```

6. Patrz 5.

7. Więcej przykładów dopasowywania argumentów. Poniżej widoczny jest rodzaj interakcji, jaki powinniśmy otrzymać, wraz z komentarzami objaśniającymi odbywające się dopasowania.

```
def f1(a, b): print(a, b)                      # Normalne argumenty

def f2(a, *b): print(a, b)                      # Zmienna liczba argumentów
pozycyjnych

def f3(a, **b): print(a, b)                      # Zmienna liczba słów
kluczowych

def f4(a, *b, **c): print(a, b, c)      # Tryby mieszane

def f5(a, b=2, c=3): print(a, b, c)      # Wartości domyślne

def f6(a, b=2, *c): print(a, b, c)          # Zmienna liczba
argumentów pozycyjnych i wartości domyślnych

% python

>>> f1(1, 2)                                # Dopasowanie po pozycji
(kolejność ma znaczenie)

1 2

>>> f1(b=2, a=1)                            # Dopasowanie po nazwie
(kolejność nie ma znaczenia)

1 2

>>> f2(1, 2, 3)                            # Dodatkowe argumenty
pozycyjne zebrane w krotkę

1 (2, 3)
```

```

>>> f3(1, x=2, y=3)                                # Dodatkowe słowa kluczowe
zebrane w słownik
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)                      # Dodatkowe argumenty obu
typów
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                                         # Obie wartości domyślne
użyte
1 2 3

>>> f5(1, 4)                                       # Jedna wartość domyślna
użyta
1 4 3

>>> f6(1)                                         # Jeden argument:
dopasowanie "a"
1 2 ( )

>>> f6(1, 3, 4)                                     # Dodatkowe argumenty
pozycyjne zebrane w krotkę
1 3 (4, )

```

8. *Powrót do liczb pierwszych.* Poniżej znajduje się przykład z liczbami pierwszymi opakowany w funkcję oraz moduł (plik *primes.py*), tak by można go było wykonać kilka razy. Dodałem test *if* przechwytyjący liczby ujemne, 0 oraz 1. Zmieniłem również operator / na //, by rozwiązywanie to było odporne na zmiany związane z „prawdziwym dzieleniem” w Pythonie 3.x (omówione w rozdziale 5.), a także by mogło ono obsługiwać liczby zmiennoprzecinkowe (aby zobaczyć różnice z wersji 2.x, należy usunąć komentarz z instrukcji *from* i zmienić operator // na /).

```

#from __future__ import division

def prime(y):
    if y <= 1:                                         # Dla jakiegoś y > 1
        print(y, 'nie jest liczbą pierwszą')
    else:
        x = y // 2                                      # 3.x / nie działa
        while x > 1:
            if y % x == 0:                               # Bez reszty?
                print(y, 'ma czynnik', x)
                break                                     # Pominięcie else
            x -= 1
        else:
            print(y, 'jest liczbą pierwszą')

prime(13); prime(13.0)

```

```
prime(15); prime(15.0)
prime(3); prime(2)
prime(1); prime(-3)
```

Poniżej widać działanie tego modułu. Operator // pozwala, by funkcja ta działała również dla liczb zmiennoprzecinkowych, nawet jeśli pewnie nie powinno tak być.

```
% python primes.py
13 jest liczbą pierwszą
13.0 jest liczbą pierwszą
15 ma czynnik 5
15.0 ma czynnik 5.0
3 jest liczbą pierwszą
2 jest liczbą pierwszą
1 nie jest liczbą pierwszą
-3 nie jest liczbą pierwszą
```

Funkcja nadal jeszcze niespecjalnie nadaje się do ponownego użycia — mogłaby zwracać wartości, zamiast je wyświetlać — jednak na potrzeby naszych eksperymentów wystarczy. Nie oblicza też liczb pierwszych w ścisłym, matematycznym znaczeniu tego pojęcia (działa na liczbach zmiennoprzecinkowych) i nadal jest niewydajna. Poprawki pozostawiam jako ćwiczenia dla osób o większych zdolnościach matematycznych (wskazówka: pętla `for` po `range(y, 1, -1)` może być nieco szybsza od instrukcji `while`, jednak algorytm jest tutaj prawdziwym wąskim gardłem). By zmierzyć alternatywne rozwiązania, należy użyć wbudowanego modułu `time`, standardowej biblioteki `timeit` i wzorców programistycznych takich jak opisane w rozdziale 21., w części poświęconej pomiarom czasu (patrz również rozwiązanie 10.).

9. *Iteratory i listy składane*. Poniżej znajduje się przykład kodu, jaki powinno się utworzyć. Być może mam swoje preferowane rozwiązanie, ale go nie zdradzę.

```
>>> values = [2, 4, 9, 16, 25]
>>> import math
>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
>>> list(map(math.sqrt, values))
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
>>> list(math.sqrt(x) for x in values)
```

```
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. *Narzędzia do pomiaru czasu.* Oto kod, który napisałem w celu pomiaru czasu trwania trzech opcji pierwiastkowania, wraz z wynikami w Pythonie 2.7, 3.3 oraz w PyPy 1.9 (który wykorzystuje Pythona 2.7). W każdym teście wybierany jest najlepszy wynik trzech prób. W każdej próbie mierzony jest całkowity czas tysiąca wywołań testowanej funkcji, natomiast każda funkcja testowa wykonuje 10 000 iteracji. Ostatni wynik każdej z funkcji jest wyświetlany w celu zweryfikowania tego, czy wszystkie trzy wykonują to samo działanie.

```
# Plik timer2.py (Python 2.x oraz 3.x)
...to samo co w rozdziale 21...

# Plik timesqrt.py

import sys, timer2

reps = 10000

repslist = range(reps)                      # Pobranie pełnego przedziału
listy pomiarów dla Pythona 2.x

from math import sqrt                         # Nie math.sqrt: dodaje czas
pobrania atrybutu

def mathMod():
    for i in repslist:
        res = sqrt(i)
    return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
    for i in repslist:
        res = i ** .5
    return res

print(sys.version)

for test in (mathMod, powCall, powExpr):
    elapsed, result = timer2.bestoftotal(test, _reps1=3,
                                          _reps=1000)
    print ('%s: %.5f => %s' %(test.__name__, elapsed, result))
```

Poniżej znajdują się wyniki testów dla trzech wersji Pythona. Wyniki uzyskane dla wersji 3.3 i 2.7 są mniej więcej dwa razy lepsze niż dla 3.0 i 3.6 z poprzedniego wydania książki. Wynika to głównie z użycia szybszego komputera. W każdej wersji Pythona moduł math wydaje się szybszy od wyrażenia z operatorem **, które z kolei jest szybsze od wywołania funkcji pow. Należy jednak spróbować samodzielnie, na własnym komputerze i zainstalowanej na nim wersji Pythona. Warto również zwrócić

uwagę na to, że Python 3.3 jest w tym teście prawie dwa razy wolniejszy od wersji 2.7, a PyPy jest mniej więcej o rząd wielkości (tj. 10 razy) szybszy od obu wersji Pythona, mimo że wszędzie wykonywane są operacje zmiennoprzecinkowe i iteracje. Nowsze wersje mogą działać lepiej (warto zmierzyć to w przyszłości samodzielnie, by się o tym przekonać).

```
C:\code> py -3 timesqrt.py
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64
bit (AMD64)]
mathMod: 2.04481 => 99.99499987499375
powCall: 3.40973 => 99.99499987499375
powExpr: 2.56458 => 99.99499987499375
C:\code> py -2 timesqrt.py
2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)]
mathMod: 1.04337 => 99.994999875
powCall: 2.57516 => 99.994999875
powExpr: 1.89560 => 99.994999875
C:\code> C:\pypy\pypy-1.9\pypy timesqrt.py
2.7.2 (341e1e3821ff, Jun 07 2012, 15:43:00)
[PyPy 1.9.0 with MSC v.1500 32 bit]
mathMod: 0.07491 => 99.994999875
powCall: 0.85678 => 99.994999875
powExpr: 0.85453 => 99.994999875
```

W celu zmierzenia względnych szybkości słowników składanych Pythona 3.x i 2.7 oraz ich odpowiedników w postaci pętli `for` w sesji interaktywnej należy wykonać sesję podobną do poniższej. Wydaje się, że oba rozwiązania są w Pythonie 3.x mniej więcej porównywalne. W przeciwieństwie do list składanych pętle wykonywane ręcznie są dziś nieco szybsze od słowników składanych (choć różnica nie jest znacząca — możemy zaoszczędzić pół sekundy, tworząc pięćdziesiąt słowników, każdy z milionem elementów). I znów, zamiast brać wyniki tego porównania za prawdę objawioną, warto sprawdzić to na własnym komputerze i zainstalowanej na nim wersji Pythona.

```
C:\code> C:\python33\python
>>>
>>> def dictcomp(I):
...     return {i: i for i in range(I)}
...
>>> def dictloop(I):
...     new = {}
...     for i in range(I): new[i] = i
...     return new
```

```

...
>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>>
>>> from timer2 import total, bestof
>>> bestof(dictcomp, 10000)[0]                                # Słownik o 10 000
elementów
0.0017095345403959072
>>> bestof(dictloop, 10000)[0]
0.002097576400046819
>>>
>>> bestof(dictcomp, 100000)[0]                                # 100 000 elementów:
10 razy wolniejszy
0.012716923463358398
>>> bestof(dictloop, 100000)[0]
0.014129806355413166
>>>
>>> bestof(dictcomp, 1000000)[0]                                # 1 000 000
elementów: 10 razy tyle czasu
0.11614425187337929
>>> bestof(dictloop, 1000000)[0]                                # Czas tworzenia
jednego słownika
0.1331144855439561
>>>
>>> timer(dictcomp, 1000000, _reps=50)[0]                      # Słownik z 1 000
000 elementów
5.8162020671780965
>>> timer(dictloop, 1000000, _reps=50)[0]                      # Czas tworzenia 50
6.626680761285343

```

11. *Funkcje rekurencyjne.* Funkcje te zakodowałem w pokazany niżej sposób. Równie dobrze mogłem wykorzystać prostą metodę `range`, wyrażenie lub metodę `map`, jednak rekurencja dobrze nadaje się do eksperymentów (`print` jest funkcją tylko w wersji 3.x, chyba że zostanie zimportowana z `__future__` lub innego, równoważnego kodu):

```

def countdown(N):
    if N == 0:

```

```

        print('stop')          # Wersja 2.x: print 'stop'
else:
    print(N, end=' ')      # Wersja 2.x: print N
    countdown(N-1)

>>> countdown(5)
5 4 3 2 1 stop
>>> countdown(20)
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 stop
# Wersje nierekurencyjne:
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
# Tylko wersja 3.x:
>>> t = [print(i, end=' ') for i in range(5, 0, -1)]
5 4 3 2 1
>>> t = list(map(lambda x: print(x, end=' '), range(5, 0, -1)))
5 4 3 2 1

```

Ze względów merytorycznych (i humanitarnych!) w powyższym kodzie nie wykorzystałem *generatora*, ale poniżej przedstawiam odpowiednie rozwiązanie. W tym przypadku wszystkie inne techniki wydają się o wiele prostsze. Jest to dobry przykład, jak można unikać stosowania generatorów. Należy pamiętać, że generator nie zwraca wyniku, dopóki nie odwołamy się do niego poprzez iterację, dlatego potrzebna jest instrukcja `for` lub `yield from` (ta druga jest dostępna tylko w wersji 3.3 lub nowszej).

```

def countdown2(N):                      # Rekurencyjna funkcja
    wykorzystująca generator
    if N == 0:
        yield 'stop'
    else:
        yield N
        for x in countdown2(N-1): yield x    # Wersja 3.3+: yield
from countdown2(N-1)

>>> list(countdown2(5))
[5, 4, 3, 2, 1, 'stop']

# Wersje nierekurencyjne:
>>> def countdown3():                  # Prostsza funkcja
    wykorzystująca generator
        yield from range(5, 0, -1)       # Wersja starsza niż
3.3: for x in range(): yield x

```

```

>>> list(countdown3())
[5, 4, 3, 2, 1]
>>> list(x for x in range(5, 0, -1))          # Wyrażenie równoważne
generatorowi
[5, 4, 3, 2, 1]
>>> list(range(5, 0, -1))                      # Równoważna forma bez
generatora
[5, 4, 3, 2, 1]

```

12. *Wyliczanie silni.* Poniżej przedstawione jest moje rozwiązywanie tego ćwiczenia. Kod działa z wersjami 3.x i 2.x. Na końcu listingu widoczny jest jego wynik w postaci literalu tekstowego w wersji 3.3. Oczywiście możliwych jest wiele odmian tego kodu. Na przykład, aby uniknąć iteracji, można wykorzystać zakresy liczb od 2 do N+1, a w funkcji fact2 zamiast wyrażenia lambda można użyć metody reduce(operator.mul, range(N, 1, -1)).

```

#!/usr/bin/python
# Plik factorials.py

from __future__ import print_function
from functools import reduce
from timeit import repeat
import math

def fact0(N):                                     # Wersja rekurencyjna
    if N == 1:                                     # Zazwyczaj ulega
        awarii dla N=999
        return N
    else:
        return N * fact0(N-1)

def fact1(N):
    return N if N == 1 else N * fact1(N-1)          # Jednowierszowa
wersja rekurencyjna

def fact2(N):                                     # Wersja funkcyjna
    return reduce(lambda x, y: x * y, range(1, N+1))

def fact3(N):
    res = 1
    for i in range(1, N+1): res *= i              # Wersja
iteracyjna
    return res

def fact4(N):

```

```

        return math.factorial(N)                      # Wersja
wykorzystująca standardową bibliotekę

# Testy

print(fact0(6),    fact1(6),    fact2(6),    fact3(6),    fact4(6))  #
6*5*4*3*2*1: all 720

print(fact0(500) == fact1(500) == fact2(500) == fact3(500) ==
fact4(500))      # True

for test in (fact0, fact1, fact2, fact3, fact4):

    print(test.__name__, min(repeat(stmt=lambda: test(500),
number=20, repeat=3)))

"""

C:\code> py -3 factorials.py

720 720 720 720 720

True

fact0 0.003990868798355564
fact1 0.003901433457907475
fact2 0.002732909419593966
fact3 0.002052614370939676
fact4 0.0003401475243271501
"""

```

Podsumowując: kod wykorzystujący rekurencję działał naj wolniej i ulegał awarii dla $N = 999$ z powodu przekroczenia domyślnej pojemności stosu określonej w module `sys`. Jak wspomniałem w rozdziale 19., stos można powiększyć, ale zdecydowanie najlepszym rozwiązaniem wydaje się zastosowanie prostej pętli lub narzędzia ze standardowej biblioteki.

Ten ogólny wniosek potwierdza się dość często. Na przykład preferowanym sposobem odwracania ciągu znaków jest użycie metody `''.join(reversed(S))`, mimo że istnieją rozwiązania rekurencyjne. Zachęcam do zmierzenia czasu wykonania poniższych funkcji. W Pythonie 3.x wyliczanie silni z wykorzystaniem rekurencji jest o rząd wielkości wolniejsze niż z użyciem innych technik, choć w języku PyPy wyniki mogą być różne.

```

def rev1(S):
    if len(S) == 1:
        return S
    else:
        return S[-1] + rev1(S[:-1])      # Wersja rekurencyjna: 10x
wolniejsza w obecnej wersji Pythona

def rev2(S):
    return ''.join(reversed(S))        # Wersja nierekurencyjna:
prostsza i szybsza

```

```
def rev3(S):
    return S[::-1]                                # Wersja jeszcze lepsza?
Odwracanie sekwencji za pomocą wycinków
```

Część V Moduły i pakiety

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części piątej” na końcu rozdziału 25.

1. *Podstawy importowania.* Kiedy skończymy, plik *mymod.py* oraz sesja interaktywna powinny wyglądać podobnie do poniższego kodu. Należy pamiętać, że Python może wczytać cały plik do listy łańcuchów znaków. Wbudowana funkcja `len` zwraca natomiast długość łańcuchów znaków oraz list.

```
def countLines(name):
    file = open(name)
    return len(file.readlines())
def countChars(name):
    return len(open(name).read())
def test(name):                                # Lub przekazanie
    obiektu pliku
    return countLines(name), countChars(name) # Lub zwrócenie
    słownika
% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291))
```

Uzyskane wyniki mogą być różne w zależności od tego, czy plik zawiera komentarze i dodatkowy wiersz na końcu. Warto zwrócić uwagę na to, że funkcje ładują cały plik w pamięci na raz, dlatego nie będą działały dla patologicznie dużych plików przekraczających możliwości pamięci komputera. Do szerszego zastosowania możemy również wczytać plik wiersz po wierszu za pomocą iteratorów i zliczać je w miarę wczytywania.

```
def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot
def countChars(name):
    tot = 0
    for line in open(name): tot += len(line)
```

```
        return tot
```

Wyrażenie generatora może dawać ten sam rezultat (jednak za nadmierną magię instruktor może zabrać punkty):

```
def countlines(name): return sum(+1 for line in open(name))
def countchars(name): return sum(len(line) for line in open(name))
```

W systemie Unix możemy zweryfikować dane wyjściowe za pomocą polecenia `wc`. W systemie Windows można kliknąć plik prawym przyciskiem myszy i sprawdzić jego właściwości. Warto jednak zauważyć, że nasz skrypt może zgłaszać mniej znaków od systemu Windows — ze względu na przenośność Python przekształca znaczniki końca wiersza `\r\n` do `\n`, usuwając tym samym jeden bajt (znak) na wiersz. By dokładnie dopasować skrypt do wyniku z systemu Windows, należy otworzyć plik w trybie binarnym ('`rb`') lub dodać liczbę bajtów odpowiadającą liczbie wierszy.Więcej informacji o przekształcaniu końców wierszy w plikach tekstowych jest zawartych w rozdziałach 9. i 37.

Aby wykonać „ambitniejszą” część ćwiczenia (przekazanie pliku obiektu, tak byśmy plik otwierali tylko raz), będziemy musieli użyć metody `seek` wbudowanego obiektu pliku. Metoda ta działa tak samo jak wywołanie `fseek` z języka C (i tak naprawdę wewnętrznie je wywołuje) — przywraca bieżącą pozycję w pliku do przekazanej wartości przesunięcia. Po wywołaniu `seek` przeszłe operacje wejścia i wyjścia odbywają się względem tej nowej pozycji. By przewinąć plik do początku bez zamykania go i ponownego otwierania, należy wywołać `file.seek(0)`. Metoda pliku `read` wznowia działanie od bieżącej pozycji w pliku, dlatego w celu ponownego wczytania go, musimy przewinąć do początku. Poniżej widać, jak mogłyby działać ta sztuczka.

```
def countLines(file):
    file.seek(0)                                              # Przewinięcie do
    początku pliku
    return len(file.readlines( ))
def countChars(file):
    file.seek(0)                                              # To samo
    (przewinięcie, jeśli jest konieczne)
    return len(file.read( ))
def test(name):
    file = open(name)                                         # Przekazanie obiektu
    pliku
    return countLines(file), countChars(file) # Plik otwierany
    tylko raz
>>> import mymod2
>>> mymod2.test("mymod2.py")
(12, 466)
```

2. *Instrukcje from i from **. Poniżej znajduje się część z `from *`. By wykonać resztę, należy zastąpić `*` za pomocą `countChars`.

```
% python
>>> from mymod import *
```

```
>>> countChars("mymod.py")
```

```
291
```

3. Wartość `__main__`. Po utworzeniu poprawnego kodu powinno to działać w obu trybach (wykonaniu programu oraz importowaniu modułu).

```
def countLines(name):  
    file = open(name)  
    return len(file.readlines())  
  
def countChars(name):  
    return len(open(name).read())  
  
def test(name):  
    # Lub przekazanie  
    # obiektu pliku  
    return countLines(name), countChars(name) # Lub zwrócenie  
    # słownika  
  
if __name__ == '__main__':  
    print(test('mymod.py'))  
  
% python mymod.py  
(13, 346)
```

W tym miejscu najprawdopodobniej zacząłbym rozważać użycie argumentów wiersza poleceń lub informacji od użytkownika w celu podania nazw plików, które mają być zliczane, zamiast zapisywać je na stałe w skrypcie. Więcej informacji na temat `sys.argv` można znaleźć w rozdziale 25., natomiast o funkcjach `input` i `raw_input` w wersji 2.x w rozdziale 10.

```
if __name__ == '__main__':  
    print(test(input('Wprowadź nazwę pliku:'))) # Konsola  
    # (raw_input w wersji 2.x)  
  
if __name__ == '__main__':  
    import sys  
    # Wiersz poleceń  
    print(test(sys.argv[1]))
```

4. Zagnieżdżone importowanie. Poniżej przedstawiam moje rozwiązanie (plik `myclient.py`).

```
from mymod import countLines, countChars  
print(countLines('mymod.py'), countChars('mymod.py'))  
  
% python myclient.py
```

```
13 346
```

Jeśli chodzi o resztę tego zadania, funkcje modułu `mymod` są dostępne (czyli można je importować) z najwyższego poziomu modułu `myclient`, ponieważ `from` po prostu przypisuje do zmiennych w pliku importującym (działa to tak samo, jakby instrukcje `def` modułu `mymod` znajdowały się w `myclient`). Przykładowo w kolejnym pliku może się znajdować taki kod:

```
import myclient
```

```

myclient.countLines(...)

from myclient import countChars

countChars(...)

```

Gdyby moduł `myclient` wykorzystał instrukcję `import` zamiast `from`, by dotrzeć do funkcji modułu `mod` za pośrednictwem `myclient`, musielibyśmy użyć ścieżki.

```

import myclient

myclient.mymod.countLines(...)

from myclient import mymod

mymod.countChars(...)

```

Zawsze możemy zdefiniować moduły *kolektorów* importujące wszystkie zmienne z innych modułów, tak by były one dostępne w jednym module. Wykorzystując poniższy kod, otrzymujemy trzy różne kopie tej samej zmiennej `somename` (`mod1.somename`, `collector.somename` oraz `__main__.somename`). Wszystkie trzy współdzielą początkowo ten sam obiekt liczby całkowitej, a tylko zmienna `somename` istnieje jako taka w sesji interaktywnej.

```

# Plik mod1.py

somename = 42

# Plik collector.py

from mod1 import *                                # Zbiera wiele
zmiennych

from mod2 import *                                # from przypisuje do
moich zmiennych

from mod3 import *

>>> from collector import somename

```

5. *Importowanie pakietów.* W celu wykonania tego ćwiczenia umieściłem rozwiązańe `mymod.py` z ćwiczenia 3. w pakiecie katalogów. Na poniższym listingu widać, co zrobiłem w interfejsie konsoli systemu Windows w celu ustawienia katalogu oraz pliku `__init__.py` wymaganego w wersjach Pythona starszych niż 3.3. Kod ten należy odpowiednio zmodyfikować na potrzeby innej platformy (na przykład użyć `cp` oraz `vi` zamiast `copy` i `notepad`). Takie rozwiązanie działa dla dowolnego katalogu (ja akurat wykorzystałem katalog ze swoimi kodami). Część tych działań można również wykonać za pomocą graficznego interfejsu użytkownika eksploratora plików.

Po skończeniu otrzymałem katalog `mypkg` zawierający pliki `__init__.py` oraz `mymod.py`. Plik `__init__.py` jest niezbędny w wersjach Pythona starszych niż 3.3 i należy go umieścić w katalogu `mypkg`, a nie w jego katalogu nadrzędnym. Katalog `mypkg` umieszczony jest w katalogu głównym podanym w ścieżce wyszukiwania modułów. Warto zwrócić uwagę na to, że instrukcja `print` umieszczona w kodzie pliku inicializującego katalogu wykonywana jest tylko przy pierwszej operacji importowania, a przy drugiej już nie. Został również wykorzystany surowy format ciągów znaków, aby uniknąć problemów ze znakami specjalnymi w ścieżkach plików.

```

C:\code> mkdir mypkg
C:\code> move mymod.py mypkg\mymod.py
C:\code> notepad mypkg\__init__.py

```

```
...zapisanie instrukcji print...
C:\code> python
>>> import mypkg.mymod
inicjalizacja mypkg
>>> mypkg.mymod.countLines(r'mypkg\mymod.py')
13
>>> from mypkg.mymod import countChars
>>> countChars(r'mypkg\mymod.py')
346
```

6. *Przeładowywanie.* Ćwiczenie to wymaga jedynie poeksperymentowania z modyfikacją pliku *changer.py* przedstawionego w książce, dlatego nie ma tu nic do pokazania.
7. *Importowanie wzajemne.* Mówiąc w skrócie, importowanie najpierw *recur2* działa, ponieważ importowanie rekurencyjne występuje wtedy przy importowaniu *recur1*, a nie przy instrukcji *from* w *recur2*.

Rozszerzone objaśnienie będzie następujące. Zimportowanie na początku *recur2* działa, ponieważ importowanie rekurencyjne z *recur1* do *recur2* pobiera *recur2* jako całość, zamiast pobierać wybrane zmienne. Moduł *recur2* jest niekompletny, kiedy importujemy go z *recur1*, jednak ponieważ wykorzystujemy instrukcję *import* zamiast *from*, jesteśmy bezpieczni — Python odnajduje i zwraca utworzony już obiekt modułu *recur2* i kontynuuje wykonywanie reszty *recur1* bez przeszkoł. Kiedy importowanie *recur2* jest wznawiane, druga instrukcja *from* odnajduje zmienną *Y* w module *recur1* (wykonanie było kompletne), dlatego nie jest zgłoszany błąd.

Wykonanie pliku jako *skryptu* nie jest tym samym co importowanie go jako modułu. Przypadki te są takie same jak wykonanie pierwszej instrukcji *import* lub *from* w sesji interaktywnej. Wykonanie *recur1* jako skryptu jest takie samo jak zimportowanie *recur2* w sesji interaktywnej, ponieważ *recur2* jest pierwszym modułem zimportowanym w *recur1*. Z tego samego powodu uruchomienie *recur2* jako skryptu nie powiedzie się — jest to to samo, co jego pierwsze interaktywne zimportowanie.

Część VI Klasy i programowanie zorientowane obiektowo

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części szóstej” na końcu rozdziału 32.

1. *Dziedziczenie.* Poniżej znajduje się kod rozwiązania dla tego ćwiczenia (plik *adder.py*) wraz z testami interaktywnymi. Metoda przeciążająca *__add__* musi się pojawić tylko raz, w klasie nadrzędnej, ponieważ wywołuje ona specyficzne dla typu metody *add* w klasach podrzędnych.

```
class Adder:
```

```

        def add(self, x, y):
            print('Nie zaimplementowano!')
        def __init__(self, start=[]):
            self.data = start
        def __add__(self, other):                      # Czy w klasach
            podrzędnych?
                return self.add(self.data, other)      # Czy zwrócić typ?
    class ListAdder(Adder):
        def add(self, x, y):
            return x + y
    class DictAdder(Adder):
        def add(self, x, y):
            new = {}
            for k in x.keys(): new[k] = x[k]
            for k in y.keys(): new[k] = y[k]
            return new
% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
Nie zaimplementowano!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}
>>> x = Adder([1])
>>> x + [2]
Nie zaimplementowano!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x

```

```
Wersja 3.3: TypeError: can only concatenate list (not "ListAdder")
to list
```

```
Wcześniejsze wersje: TypeError: __add__ nor __radd__ defined for
these operands
```

Warto zauważyc, że w ostatnim teście otrzymujemy błąd dla wyrażeń, w których instancja klasy pojawia się po prawej stronie operatora +. Jeśli chcemy to naprawić, musimy użyć metod `__radd__`, zgodnie z opisem z rozdziału 30., „Przeciążanie operatorów”.

Jeśli w jakiś sposób zapisujemy wartość instancji, możemy również dobrze przepisać metodę `add` w taki sposób, by przyjmowała tylko jeden argument, zgodnie z innymi przykładami z tej części książki (patrz plik `adder2.py`).

```
class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                      # Przekazanie
                                                # pojedynczego argumentu
        return self.add(other)                     # Lewa strona jest w
self
    def add(self, y):
        print('Nie zaimplementowano!')
class ListAdder(Adder):
    def add(self, y):
        return self.data + y
class DictAdder(Adder):
    def add(self, y):
        d = self.data.copy()                      # Zmodyfikuj, by użyć
self.data zamiast x
        d.update(y)
        return d
x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)                                         # Wyświetla [1, 2, 3,
4, 5, 6]
z = DictAdder(dict(name='Robert')) + {'a':1}
print(z)                                         # Wyświetla {'name':
'Robert', 'a': 1}
```

Ponieważ wartości są raczej dołączane do obiektów niż przekazywane, ta wersja jest bardziej zorientowana obiektywnie. A skoro już doszliśmy aż tutaj, najprawdopodobniej okaże się, że możemy się całkowicie pozbyć metody `add` i po prostu zdefiniować specyficzne metody `__add__` w dwóch klasach podanych.

2. *Przeciążanie operatorów*. Kod rozwiązań (plik *mylist.py*) wykorzystuje kilka opisanych w rozdziale 30. metod przeciążania operatorów. Skopiowanie wartości początkowej w konstruktorze jest ważne, ponieważ może ona być zmienna. Nie chcemy modyfikować lub mieć referencji do obiektu, który może być współdzielony gdzieś poza klasą. Metoda `__getattr__` przekierowuje wywołania do opakowanej listy. Wskazówki dotyczące łatwiejszego sposobu zapisania tego kodu w Pythonie 2.2 oraz nowszych wersjach można znaleźć w podrozdziale „Rozszerzanie typów za pomocą klas podrzędnych” w rozdziale 32.

```
class MyList:

    def __init__(self, start):
        #self.wrapped = start[:]                                # Skopiowanie start –
        brak efektów ubocznych

        self.wrapped = list(start)                            # Upewniamy się, że
        jest to lista

    def __add__(self, other):
        return MyList(self.wrapped + other)

    def __mul__(self, time):
        return MyList(self.wrapped * time)

    def __getitem__(self, offset):                         # W wersji 3.x
        przekazane również jako wycinek
        return self.wrapped[offset]                        # Iterowanie, jeżeli
        brak __iter__

    def __len__(self):
        return len(self.wrapped)

    def __getslice__(self, low, high):                   # Ignorowane w wersji
        3.x: wykorzystuje __getitem__
        return MyList(self.wrapped[low:high])

    def append(self, node):
        self.wrapped.append(node)

    def __getattr__(self, name):                          # Inne metody: sort,
        reverse itd.
        return getattr(self.wrapped, name)

    def __repr__(self):                                 # Metoda
        wyświetlająca wszystkie dane
        return repr(self.wrapped)

if __name__ == '__main__':
    x = MyList('mielonka')
    print(x)
    print(x[2])
    print(x[1:])
```

```

    print(x + ['jajka'])
    print(x * 3)
    x.append('a')
    x.sort()
    print(' '.join(c for c in x))

C:\code> python mylist.py
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a']
e
['i', 'e', 'l', 'o', 'n', 'k', 'a']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'jajka']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'm', 'i', 'e', 'l', 'o',
'n', 'k', 'a', 'm', 'i', 'e', 'l', 'o', 'n', 'k', 'a']

a a e i k l m n o

```

Warto zauważyc, że bardzo istotne jest skopiowane wartości początkowej za pomocą dodania do listy, a nie wycinka, ponieważ w innym przypadku wynik mógłby nie być prawdziwą listą i tym samym nie reagowałby na oczekiwane metody listy, takie jak `append` (wycinek łańcucha znaków zwraca kolejny łańcuch znaków, a nie listę). Bylibyśmy w stanie skopiować wartość początkową `MyList` za pomocą wycinka, ponieważ jej klasa przeciąża operację tworzenia wycinka i udostępnia oczekiwany interfejs listy. Musimy jednak unikać kopowania opartego na wycinkach dla obiektów takich, jak łańcuchy znaków.

3. *Klasy podrzędne.* Moja propozycja rozwiązania (plik `mysub.py`) znajduje się poniżej. Rozwiązanie tego ćwiczenia powinno być podobne do poniższego.

```

from mylist import MyList

class MyListSub(MyList):
    calls = 0                                         # Współdzielona przez
    instancje

    def __init__(self, start):
        self.adds = 0                                     # Różni się w każdej
        instancji
        MyList.__init__(self, start)

    def __add__(self, other):
        print('dodane: ' + str(other))
        MyListSub.calls += 1                            # Licznik dla całej
        klasy
        self.adds += 1                                  # Liczniki dla
        instancji
        return MyList.__add__(self, other)

    def stats(self):

```

```

        return self.calls, self.adds           # Wszystkie
dodawania, moje dodawania

if __name__ == '__main__':
    x = MyListSub('mielonka')
    y = MyListSub('szynka')
    print(x[2])
    print(x[1:])
    print(x + ['jajka'])
    print(x + ['tost'])
    print(y + ['bar'])
    print(x.stats( ))

```

C:\code> **python mysub.py**

```

e
['i', 'e', 'l', 'o', 'n', 'k', 'a']
dodane: ['jajka']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'jajka']
dodane: ['tost']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'tost']
dodane: ['bar']
['s', 'z', 'y', 'n', 'k', 'a', 'bar']
(3, 2)

```

4. *Metody atrybutów.* Ja rozwiązałem to ćwiczenie w następujący sposób. Warto zauważyc, że w Pythonie 2.x operatory próbują również pobrać atrybuty za pomocą metody `__getattr__`. Musimy zwrócić uwagę, by działały. Uwaga: zgodnie z informacjami z rozdziału 32. metoda `__getattr__` nie jest w Pythonie 3.x (jak również w 2.x, jeżeli stosowane są klasy w nowym stylu) wywoływana dla operacji wbudowanych, dlatego wyrażenia nie są w ogóle przechwytywane. W wersji 3.x klasa w nowym stylu, taka jak ta, musi redefiniować metodę przeciążania operatora `_X_` w sposób jawnego.Więcej informacji na ten temat można znaleźć w rozdziałach 28., 31., 32., 38. oraz 39. Zmiany te mogą mieć duży wpływ na kod!

```

C:\code> py -2
>>> class Attrs:
...     def __getattr__(self, name):
...         print('pobierz %s' % name)
...     def __setattr__(self, name, value):
...         print('ustaw %s %s' % (name, value))
...
>>> x = Attrs( )

```

```

>>> x.append
pobierz append
>>> x.spam = "wieprzowina"
ustaw spam wieprzowina
>>> x + 2
pobierz __coerce__
TypeError: 'NoneType' object is not callable
>>> x[1]
pobierz __getitem__
TypeError: 'NoneType' object is not callable
>>> x[1:5]
pobierz __getslice__
TypeError: 'NoneType' object is not callable
C:\code> py -3
>>> ...ten sam kod startowy...
>>> x + 2
TypeError: unsupported operand type(s) for +: 'Attrs' and 'int'
>>> x[1]
TypeError: 'Attrs' object does not support indexing
>>> x[1:5]
TypeError: 'Attrs' object is not subscriptable

```

5. *Obiekty zbiorów.* Poniżej znajduje się rodzaj sesji interaktywnej, jaką powinniśmy uzyskać. Komentarze wyjaśniają, która metoda jest wywoływana. Warto pamiętać, że w obecnej wersji Pythona zbiór jest typem wbudowanym, więc jest to przede wszystkim ćwiczenie z kodowania (więcej informacji na temat zbiorów zawiera rozdział 5.).

```

C:\code> python
>>> from setwrapper import Set
>>> x = Set([1, 2, 3, 4])                                # Wykonuje __init__
>>> y = Set([3, 4, 5])
>>> x & y                                              # __and__, intersect,
następnie __repr__
Zbiór:[3, 4]
>>> x | y                                              # __or__, union,
następnie __repr__
Zbiór:[1, 2, 3, 4, 5]

```

```

>>> z = Set("hallo")                                # __init__ usuwa
duplikaty
>>> z[0], z[-1]                                    # __getitem__
('h', 'o')
>>> for c in z: print(c, end=' ')                 # __iter__ (lub
__getitem__) [w wersji 3.x print]
...
h a l o
>>> ''.join(c.upper() for c in z)                # __iter__ (lub
__getitem__)
'HALO'
>>> len(z), z                                     # __len__, __repr__
(4, Zbiór:['h', 'a', 'l', 'o'])
>>> z & "mallو", z | "mallو"
(Zbiór:['a', 'l', 'o'], Zbiór:['h', 'a', 'l', 'o', 'm'])

```

Moje rozwiązanie klasy podrzędnej rozszerzenia z wieloma argumentami przypomina klasę zaprezentowaną niżej (plik *multiset.py*). Wystarczy jedynie zastąpić dwie metody w oryginalnym zbiorze. Łańcuch znaków dokumentacji klasy wyjaśnia, jak to działa.

```

from setwrapper import Set
class MultiSet(Set):
    """
    Dziedziczy wszystkie zmienne klasy Set, jednak
    rozszerza intersect oraz union w taki sposób, by
    obsługiwały większą liczbę argumentów. Warto
    zauważyc, że "self" nadal jest pierwszym
    argumentem (przechowanym teraz w argumencie *args).
    Odziedziczone operatory & oraz | wywołują nowe
    metody z dwoma argumentami, jednak przetworzenie
    więcej niż dwóch wymaga wywołania metody, a nie
    wyrażenia. Metoda intersect nie usuwa duplikatów –
    robi to konstruktor Set.
    """
    def intersect(self, *others):
        res = []
        for x in self:                                # Przeszukanie
            pierwszej sekwencji

```

```

        for other in others:                      # Dla wszystkich
pozostałych argumentów

            if x not in other: break             # Element w każdym z
nich?

        else:                                # Nie – wyjście z
pętli

            res.append(x)                      # Tak – dodanie
elementu na końcu

    return Set(res)

def union(*args):                         # self to args[0]

    res = []
    for seq in args:                      # Dla wszystkich
argumentów

        for x in seq:                      # Dla wszystkich
węzłów

            if not x in res:
                res.append(x)              # Dodanie nowych
elementów do wyniku

    return Set(res)

```

Nasza interakcja z tym rozszerzeniem będzie przypominała poniższą. Warto zauważyć, że możemy obliczać część wspólną zbiorów za pomocą & lub wywołania `intersect`, natomiast dla trzech lub większej liczby argumentów musimy już wywołać `intersect`. Operator & jest operatorem binarnym (dwustronnym). Moglibyśmy również po prostu `Multiset` nazwać `Set`, by zmiana ta była bardziej niewidoczna, gdybyśmy użyli `setwrapper.Set` do odniesienia się do oryginału wewnętrz klasy `multiset` (w razie potrzeby można użyć klauzuli `as` w instrukcji `import`, aby zmienić nazwę klasy).

```

>>> from multiset import *
>>> x = MultiSet([1,2,3,4])
>>> y = MultiSet([3,4,5])
>>> z = MultiSet([0,1,2])
>>> x & y, x | y                      # Dwa operandy
(Zbiór:[3, 4], Zbiór:[1, 2, 3, 4, 5])
>>> x.intersect(y, z)                  # Trzy argumenty
Zbiór:[]
>>> x.union(y, z)
Zbiór:[1, 2, 3, 4, 5, 0]
>>> x.intersect([1,2,3], [2,3,4], [1,2,3])  # Cztery argumenty
Zbiór:[2, 3]

```

```

>>> x.union(range(10))                                # Działa również na
      innych zbiorach
Zbiór:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]

>>> w = MultiSet('mielonka')
>>> w
Zbiór:['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a']

>>> ''.join(w | 'super')
'mielonkasupr'

>>> (w | 'super') & MultiSet('sloty')
Zbiór:['l', 'o', 's']

```

6. Łącza drzew klas. Poniżej widać sposób, w jaki ja zmodyfikowałem klasę Lister, a także powtóżenie testu pokazującego jej format. W przypadku wersji opartej na funkcji dir należy zrobić to samo; podobnie będzie w przypadku formatowania obiektów klas w wariancie z przechodzienniem drzewa klas.

```

class ListInstance:

    def __attrnames(self):
        ...bez zmian...

    def __str__(self):
        return '<Instancja klasy %s(%s), adres %s:\n%s>' % (
            self.__class__.__name__,      # Nazwa mojej klasy
            self.__supers(),             # Moje klasy
            nadziedne
            id(self),                  # Mój adres
            self.__attrnames() )        # Lista nazwa=wartość

    def __supers(self):
        names = []
        for super in self.__class__.__bases__: # Jeden poziom w góre
od klasy
            names.append(super.__name__)          # name, nie
str(super)

        return ', '.join(names)
            # Lub: ', '.join(super.__name__ for super in
self.__class__.__bases__)

C:\code> python listinstance-exercise.py
<Instancja klasy Sub(Super, ListInstance), adres 7841200:
zmienna data1=mielonka
zmienna data2=jajka
zmienna data3=42

```

>

7. *Kompozycja*. Moja propozycja rozwiązania (plik *lunch.py*) znajduje się poniżej, wraz z komentarzami z opisem umieszczonymi w kod. W tym przypadku wydaje się, że łatwiej jest problem opisać w Pythonie niż w języku polskim.

```
class Lunch:  
    def __init__(self):  
        # Utworzenie/osadzenie klas Customer oraz Employee  
        self.cust = Customer()  
        self.empl = Employee()  
  
    def order(self, foodName):  
        # Rozpoczęcie symulacji zamówienia klienta Customer  
        self.cust.placeOrder(foodName, self.empl)  
  
    def result(self):  
        # Zapytanie klienta Customer o jego jedzenie Food  
        self.cust.printFood()  
  
class Customer:  
    def __init__(self):  
        # Inicjalizacja mojego jedzenia na None  
        self.food = None  
  
    def placeOrder(self, foodName, employee):  
        # Złożenie zamówienia pracownikowi Employee  
        self.food = employee.takeOrder(foodName)  
  
    def printFood(self):  
        # Wyświetlenie nazwy mojego jedzenia  
        print(self.food.name)  
  
class Employee:  
    def takeOrder(self, foodName):  
        # Zwrócenie jedzenia Food o żądanej nazwie  
        return Food(foodName)  
  
class Food:  
    def __init__(self, name):  
        # Przechowanie nazwy jedzenia  
        self.name = name  
  
    if __name__ == '__main__':  
        x = Lunch()  
        # Kod samosprawdzający  
        x.order('burrito')  
        # Jeśli plik jest wykonywany, nie importowany  
        x.result()
```

```

        x.order('pizza')
        x.result( )
C:\code> python lunch.py
burrito
pizza

```

8. *Hierarchia zwierząt w zoo.* Poniżej znajduje się sposób, w jaki ja utworzyłem taksonomię zwierząt w kodzie napisanym w Pythonie (plik *zoo.py*). Jest ona sztuczna, jednak ten ogólny wzorzec kodu ma zastosowanie do wielu prawdziwych struktur, od graficznych interfejsów użytkownika po bazy danych pracowników. Warto zauważyć, że referencja `self.speak` w `Animal` uruchamia niezależne wyszukiwanie dziedziczenia, które odnajduje metodę `speak` w klasie podzielonej. Należy przetestować ten kod interaktywnie zgodnie z opisem z ćwiczenia. Można spróbować rozszerzyć tę hierarchię za pomocą nowych klas, a także utworzyć instancje poszczególnych klas drzewa.

```

class Animal:
    def reply(self): self.speak( )                      # Z powrotem do klasy
    podzielonej

    def speak(self): print('mielonka')                  # Własny komunikat

class Mammal(Animal):
    def speak(self): print('he?')

class Cat(Mammal):
    def speak(self): print('miau')

class Dog(Mammal):
    def speak(self): print('hau')

class Primate(Mammal):
    def speak(self): print('Witaj, świecie! ')

class Hacker(Primate): pass                          # Dziedziczy po
    klasie Primate

```

9. *Skecz z martwą papugą.* Poniżej widać, jak ja zaimplementowałem to zadanie (plik *parrot.py*). Warto zwrócić uwagę na to, jak działa metoda `line` w klasie nadzielonej `Actor`. Dzięki dwukrotnemu dostępowi do atrybutów `self` dwa razy odsyła ona Pythona z powrotem do instancji i tym samym uruchamia dwa wyszukiwania dziedziczenia — `self.name` oraz `self.says()` odnajdują informacje w określonych klasach podzielonych.

```

class Actor:
    def line(self): print(self.name + ':', repr(self.says( )))

class Customer(Actor):
    name = 'klient'

    def says(self): return "To już ekspapuga!"

class Clerk(Actor):

```

```

        name = 'sprzedawca'

    def says(self): return "nie, wcale nie..."

class Parrot(Actor):
    name = 'papuga'

    def says(self): return None

class Scene:
    def __init__(self):
        self.clerk = Clerk( )                      # Osadzenie instancji
        self.customer = Customer( )                # Scene to kompozyt
        self.subject = Parrot( )

    def action(self):
        self.customer.line( )                     # Delegacja do
osadzonych
        self.clerk.line( )
        self.subject.line( )

```

Część VII Wyjątki oraz narzędzia

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części siódmej” na końcu rozdziału 36.

1. *Instrukcja try/except.* Moja wersja funkcji oops (plik *oops.py*) znajduje się poniżej. Jeśli chodzi o pytania dodatkowe, modyfikacja oops w taki sposób, by funkcja ta zgłaszała wyjątek `KeyError` zamiast `IndexError` oznacza, że program obsługi nie będzie przechwytywał wyjątku (zostanie on przekazany na najwyższy poziom programu i wywoła domyślny komunikat o błędzie Pythona). Nazwy `KeyError` oraz `IndexError` pochodzą z najbardziej zewnętrznego zakresu wbudowanego. By to zobaczyć, należy zainportować moduł `__builtins__` (`__builtin__` w Pythonie 2.x) i przekazać go jako argument do funkcji `dir`.

```

def oops( ):
    raise IndexError

def doomed( ):
    try:
        oops( )
    except IndexError:
        print('przechwycono błąd indeksu!')
    else:
        print('nie przechwycono żadnego błędu... ')

```

```

if __name__ == '__main__': doomed( )
C:\code> python oops.py
przechwycono błąd indeksu!

```

2. *Obiekty wyjątków oraz listy.* Poniżej widać mój sposób rozszerzenia tego modułu, tak by obsługiwał on mój własny wyjątek (plik *oops2.py*):

```

from __future__ import print_function # Wersja 2.x
class MyError(Exception): pass
def oops( ):
    raise MyError('Mielonka!')
def doomed( ):
    try:
        oops( )
    except IndexError:
        print('przechwycono błąd indeksu!')
    except MyError as data:
        print('przechwycono błąd:', MyError, data)
    else:
        print('nie przechwycono żadnego błędu...')
if __name__ == '__main__':
    doomed( )
C:\code> python oops.py
przechwycono błąd: <class '__main__.MyError'> Mielonka!

```

Tak jak w przypadku wszystkich innych wyjątków opartych na klasach, instancja ta jest dostępna przez zmienną `data`. Komunikat o błędzie pokazuje teraz zarówno klasę (`<...>`), jak i jejinstancję (`Mielonka!`). Instancja ta musi dziedziczyć zarówno metodę `__init__`, jak i `__repr__` bądź `__str__` po klasie Pythona o nazwie `Exception` — w przeciwnym razie wyświetlana byłaby tak samo jak klasa. Więcej informacji na temat tego, jak działa to dla wbudowanych klas wyjątków, można znaleźć w rozdziale 35.

3. *Obsługa błędów.* Poniżej znajduje się jeden ze sposobów wykonania tego ćwiczenia (plik *exctools.py*). Swoje testy przeprowadzałem w pliku, a nie w sesji interaktywnej, ale wyniki powinny być podobne. Warto zwrócić uwagę, że użycie pustej instrukcji `except` oraz metody `sys.exc_info` powoduje przechwytywanie wyjątków powodujących wyjście z programu, co nie jest możliwe za pomocą klasy `Exception` i instrukcji `as`. Prawdopodobnie w większości aplikacji nie będzie to idealne rozwiązanie, ale może stanowić przydatne narzędzie pełniące rolę swego rodzaju zapory dla wyjątków.

```

import sys, traceback
def safe(callee, *pargs, **kargs):
    try:

```

```

        callee(*pargs, **kargs)           #
Przechwycenie wszystkiego innego

    except:
        traceback.print_exc( )
        print('Mam %s %s' % ( sys.exc_info()[0], sys.exc_info()[1]))

if __name__ == '__main__':
    import oops
    safe(oops.oops)

C:\code> py -3 exctools.py
Traceback (most recent call last):
  File "C:\code\exctools.py", line 5, in safe
      callee(*pargs, **kargs)           #
Przechwycenie wszystkiego innego
  File "C:\code\oops.py", line 6, in oops
      raise MyError('Mielonka!')
oops2.MyError: Mielonka!
Mam <class 'oops2.MyError'> Mielonka!

```

Przy użyciu technik przedstawionych w rozdziale 32. i opisanych dokładniej w rozdziale 39. można powyższy kod przekształcić w dekorator przechwytyjący wyjątki zgłaszone przez dowolne funkcje. Taki dekorator rozszerza funkcję, dzięki czemu nie trzeba jej jawnie przekazywać:

```

import sys, traceback

def safe(callee):
    def callproxy(*pargs, **kargs):
        try:
            return callee(*pargs, **kargs)
        except:
            traceback.print_exc()
            print('Mam %s %s' % (sys.exc_info()[0], sys.exc_info()[1]))
            raise
    return callproxy

if __name__ == '__main__':
    import oops2
    @safe
    def test():
        oops2.oops()

```

```
    test()
```

4. Przykłady do samodzielnego przestudiowania. Poniżej znajduje się kilka przykładów kodu, które można przestudiować w miarę posiadania chwili wolnego czasu. Więcej podobnych przykładów znajduje się w innych książkach (np. *Programming Python*, z której pochodzą przykłady przedstawione tutaj) oraz w internecie.

```
# Odnalezienie największego pliku źródłowego Pythona w jednym katalogu

import os, glob

dirname = r'C:\Python33\Lib'

allsizes = []

allpy = glob.glob(dirname + os.sep + '*.py')

for filename in allpy:

    filesize = os.path.getsize(filename)

    allsizes.append((filesize, filename))

allsizes.sort()

print(allsizes[:2])

print(allsizes[-2:])

# Odnalezienie największego pliku źródłowego Pythona w całym drzewie katalogów

import sys, os, pprint

if sys.platform[:3] == 'win':

    dirname = r'C:\Python33\Lib'

else:

    dirname = '/usr/lib/python'

allsizes = []

for (thisDir, subshere, fileshere) in os.walk(dirname):

    for filename in fileshere:

        if filename.endswith('.py'):

            fullname = os.path.join(thisDir, filename)

            fullsize = os.path.getsize(fullname)

            allsizes.append((fullsize, fullname))

allsizes.sort()

pprint.pprint(allsizes[:2])

pprint.pprint(allsizes[-2:])

# Odnalezienie największego pliku źródłowego Pythona w ścieżce wyszukiwania

# importowanych modułów
```

```
import sys, os, pprint
visited = {}
allsizes = []
for srccdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srccdir):
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('pomijam', pypath)
                allsizes.append((pysize, pypath))
allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])
# Sumowanie rozdzielonych przecinkami kolumn w pliku tekstowym
filename = 'data.txt'
sums = {}
for line in open(filename):
    cols = line.split(',')
    nums = [int(col) for col in cols]
    for (ix, num) in enumerate(nums):
        sums[ix] = sums.get(ix, 0) + num
for key in sorted(sums):
    print(key, '=', sums[key])
# Podobnie do poprzedniego przykładu, jednak dla sum wykorzystuje
listy,
# a nie słowniki
import sys
```

```

filename = sys.argv[1]
numcols = int(sys.argv[2])
totals = [0] * numcols
for line in open(filename):
    cols = line.split(',')
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums)]
print(totals)
# Testowanie regresji w danych wyjściowych zbioru skryptów
import os
testscripts = [dict(script='test1.py', args=''),           # Lub
               dict(script='test2.py', args='spam')]
for testcase in testscripts:
    commandline = '%(script)s %(args)s' % testcase
    output = os.popen(commandline).read( )
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Utworzony:', result)
    else:
        priorresult = open(result).read( )
        if output != priorresult:
            print('NIEPOWODZENIE:', testcase['script'])
            print(output)
        else:
            print('Przeszedł:', testcase['script'])
# Utworzenie graficznego interfejsu użytkownika za pomocą tkinter
# (w Pythonie 2.x Tkinter) z przyciskami zmieniającymi kolor oraz
# wielkość
from tkinter import *                                         # W
Pythonie 2.x należy użyć Tkinter
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'white',
          'cyan', 'purple']

```

```

def reply(text):
    print(text)
    popup = Toplevel( )
    color = random.choice(colors)
    Label(popup,    text='Okno wyskakujące',    bg='black',
fg=color).pack( )
    L.config(fg=color)
def timer( ):
    L.config(fg=random.choice(colors))
    win.after(250, timer)
def grow( ):
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)
win = Tk( )
L = Label(win, text='Mielonka',
          font=('arial', fontsize, 'italic'), fg='yellow',
          bg='navy', relief=RAISED)
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win,           text='Naciśnij',           command=(lambda:
reply('red'))).pack(side=BOTTOM, fill=X)
Button(win,   text='Licznik',   command=timer).pack(side=BOTTOM,
fill=X)
Button(win,   text='Powiększ' ,   command=grow).pack(side=BOTTOM,
fill=X)
win.mainloop( )
# Podobnie do poprzedniego przykładu, ale wykorzystuje klasy, tak
by każde okno
# miało własne informacje o stanie
from tkinter import *
import random
class MyGui:
    """
    Graficzny interfejs użytkownika z przyciskami zmieniającymi
    kolor i powiększającymi napis
    """

```

```

colors = ['blue', 'green', 'orange', 'red', 'brown', 'yellow']
def __init__(self, parent, title='Okno wyskakujące'):
    parent.title(title)
    self.growing = False
    self.fontsize = 10
    self.lab = Label(parent, text='Guil', fg='white', bg='navy')
    self.lab.pack(expand=YES, fill=BOTH)

        Button(parent,      text='Mielonka',
command=self.reply).pack(side=LEFT)

        Button(parent,      text='Powiększ',
command=self.grow).pack(side=LEFT)

        Button(parent,      text='Zatrzymaj',
command=self.stop).pack(side=LEFT)

    def reply(self):
        "losowa modyfikacja koloru przycisku po naciśnięciu
przycisku Mielonka"
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color, font=('courier', self.fontsize,
'bold italic'))

    def grow(self):
        "po naciśnięciu przycisku Powiększ podpis zaczyna rosnąć"
        self.growing = True
        self.grower()

    def grower(self):
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fontsize, 'bold'))
            self.lab.after(500, self.grower)

    def stop(self):
        "po naciśnięciu przycisku Zatrzymaj przycisk przestaje
rosnąć"
        self.growing = False

class MySubGui(MyGui):
    colors = ['black', 'purple']          # Można dostosować w celu
modyfikacji kolorów do wyboru
    MyGui(Tk( ), 'main')

```

```

MyGui(Toplevel( ))
MySubGui(Toplevel( ))
mainloop( )
# Narzędzie do przeglądania folderu poczty przychodzącej oraz
utrzymywania jej
"""
przegląda skrzynkę e-mailową POP, pobierając tylko
nagłówki i pozwalając na usuwanie bez pobierania pełnej wiadomości
"""

import poplib, getpass, sys
mailserver = 'nazwa serwera POP naszej poczty' # pop.rmi.net
mailuser = 'identyfikator użytkownika poczty' # brian
mailpasswd = getpass.getpass('Hasło dla %s?' % mailserver)
print('Łączenie...')
server = poplib.POP3(mailserver)
server.user(mailuser)
server.pass_(mailpasswd)
try:
    print(server.getwelcome())
    msgCount, mboxSize = server.stat()
    print('W skrzynce jest', msgCount, 'wiadomości e-mail, rozmiar',
          ', mboxSize)
    msginfo = server.list()
    print(msginfo)
    for i in range(msgCount):
        msgnum = i+1
        msgsize = msginfo[1][i].split()[1]
        resp, hdrlines, octets = server.top(msgnum, 0) # Pobranie tylko nagłówków
        print('-'*80)
        print('[%d: octets=%d, size=%s]' % (msgnum, octets,
msgsize))
        for line in hdrlines: print(line)
        if input('Wyświetlić?') in ['y', 'Y']:
            for line in server.retr(msgnum)[1]: print(line) # Pobranie całej wiadomości

```

```

        if input('Usunąć?') in ['y', 'Y']:
            print('usuwanie')
            server.delete(msgnum)                                #
            Usunięcie z serwera

        else:
            print('pomijanie')

    finally:
        server.quit()                                     # Należy
        odblokować skrzynkę mbox

    input('Żegnaj.')                                    #
    Zachowanie okna

# Skrypt CGI po stronie klienta wchodzący w interakcję z
# przeglądarką

#!/usr/bin/python

import cgi

form = cgi.FieldStorage()                           # Przetworzenie
danych formularza

print("Content-type: text/html\n")                  # Nagłówek i pusty
wiersz

print("<HTML>")

print("<title>Strona odpowiedzi</title>")        # Strona html z
odpowiedzią

print("<BODY>")

if not 'user' in form:
    print("<h1>Kim jesteś?</h1>")
else:
    print("<h1>Witaj,      <i>%s</i>!</h1>" %
cgi.escape(form['user'].value))

print("</BODY></HTML>")

# Skrypt bazy danych wypełniający obiekt shelve obiektami Pythona

# Zachęcam do zjrzenia również do przykładów z rozdziału 28.
(shelve) oraz z rozdziału 31. (pickle)

rec1 = {'name': {'first': 'Robert', 'last': 'Kowalski'},
        'job': ['programista', 'menedżer'],
        'age': 40.5}

rec2 = {'name': {'first': 'Zuzanna', 'last': 'Zielona'},
        'job': ['menedżer'],
        'age': 35.0}

```

```
import shelve
db = shelve.open('dbfile')
db['bob'] = rec1
db['sue'] = rec2
db.close( )
# Skrypt bazy danych wyświetlający i aktualniający obiekt shelve
utworzony
# w poprzednim skrypcie
import shelve
db = shelve.open('dbfile')
for key in db:
    print(key, '=>', db[key])
bob = db['bob']
bob['age'] += 1
db['bob'] = bob
db.close( )
# Skrypt bazy danych zapełniający bazę MySQL i wykonujący do niej
zapytania
from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='XXXXXXXX')
curs = conn.cursor( )
try:
    curs.execute('drop database testpeopledb')
except:
    pass                                     # Nie istniała
curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10),
pay int(4))')
curs.execute('insert people values (%s, %s, %s)', ('Robert',
'programista', 50000))
curs.execute('insert people values (%s, %s, %s)', ('Zuzanna',
'programista', 60000))
curs.execute('insert people values (%s, %s, %s)', ('Anna',
'menedżer', 40000))
curs.execute('select * from people')
for row in curs.fetchall():
```

```

        print(row)

curs.execute('select * from people where name = %s', ('Robert',))
print(curs.description)
colnames = [desc[0] for desc in curs.description]
while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
    for (name, value) in zip(colnames, row):
        print('%s => %s' % (name, value))

conn.commit()                                     # Zapisanie
wstawionych rekordów

# Pobranie pliku z serwera FTP i otworzenie go
import webbrowser, sys

from ftplib import FTP                           # Narzędzia FTP
oparte na gniazdach

from getpass import getpass                      # Poufne
wprowadzanie hasła

if sys.version[0] == '2': input = raw_input      # Kompatybilność z
wersją 2.x

nonpassive = False                                # Wymusić tryb
aktywny FTP?

filename = input('Plik?')                         # Nazwa pobieranego
pliku

dirname = input('Katalog? ') or '.'                # Katalog źródłowy
na serwerze

sitename = input('Adres?')                        # Adres serwera FTP

user = input('Login?')                            # Wpisz (), aby
nawiązać połączenie anonimowe

if not user:
    userinfo = ()

else:
    from getpass import getpass                  # Poufne
wprowadzanie hasła

    userinfo = (user, getpass('Hasło?'))

print('Łączenie...')

connection = FTP(sitename)                      # Połączenie z
serwerem FTP

```

```
connection.login(*userinfo)                      # Domyślny tryb
anonimowy

connection.cwd(dirname)                         # Pobieranie pliku w
porcjach po 1 kB

if nonpassive:                                  # Włączenie trybu
aktywnego na żądanie serwera

    connection.set_pasv(False)

print('Pobieranie...')

localfile = open(filename, 'wb')                  # Zapisanie
pobranego pliku na lokalnym dysku

connection.retrbinary('RETR ' + filename, localfile.write, 1024)

connection.quit()

localfile.close()

print('Otwieranie...')

webbrowser.open(filename)
```

O autorze

Mark Lutz jest znanym na całym świecie instruktorem Pythona, autorem najwcześniejszych i najlepiej się sprzedających tekstu poświęconych temu językowi oraz jedną z najważniejszych postaci w środowisku Pythona.

Mark jest również autorem popularnych książek: *Programming Python* (O'Reilly) oraz *Python. Leksykon kieszonkowy* (wydanie polskie: Helion) — obie doczekały się już czwartego bądź piątego wydania. Używa i promuje Pythona od 1992 roku, zaczął pisać książki na temat tego języka w 1995 roku, szkolenia z Pythona prowadzi od 1997 roku i do lata 2015 r. przeprowadził ponad dwieście sześćdziesiąt poświęconych temu językowi sesji treningowych dla około czterech tysięcy osób. Jego książki poświęcone Pythonowi sprzedawały się w nakładzie około pięciuset tysięcy egzemplarzy i zostały przetłumaczone na kilkanaście języków.

Dwie dekady pracy autora przyczyniły się do tego, że Python jest dzisiaj jednym z najpopularniejszych języków programowania. Mark jest związany z informatyką od 30 lat. Jest absolwentem tego kierunku na University of Wisconsin, gdzie zajmował się implementacją języka Prolog i jako specjalista od programowania pracował nad kompilatorami, narzędziami programistycznymi, aplikacjami skryptowymi oraz wybranymi systemami klient-serwer.

Mark prowadzi stronę internetową zawierającą materiały szkoleniowe (<http://learning-python.com/training>) oraz informacje uzupełniające do jego książek (<http://learning-python.com/books>).

Kolofon

Zwierzę na okładce książki *Python. Wprowadzenie. Wydanie V* to amerykański szczur drzewny z rodzaju *Neotoma muridae*. Mieszka w różnych warunkach środowiskowych (przede wszystkim na obszarach skalistych, pustynnych lub w buszu) na większości terytorium Ameryki Północnej oraz Środkowej, zazwyczaj w pewnym oddaleniu od ludzi. Szczyrki te dobrze się wspinają, a gniazda zakładają w drzewach lub zaroślach na wysokości do sześciu metrów. Niektóre gatunki kopią podziemne nory lub mieszkają w skalnych szczelinach bądź zajmują nory opuszczone przez inne zwierzęta.

Te szarobeżowe gryzonie średniej wielkości są prawdziwymi „chomikami” — znoszą do swoich siedzib dokładnie wszystko, obojętnie, czy jest im potrzebne, czy nie, i są szczególnymi wielbicielami świecących przedmiotów, takich jak puszki, szkło czy srebro.

Rysunek na okładce to dziewiętnastowieczny sztych z *Cuvier's Animals*.

Spis treści

Przedmowa
„Ekosystem” tej książki
O niniejszym, piątym wydaniu książki
Python 2.x i 3.x kiedyś
Python 2.x i 3.x obecnie
Omawiamy zarówno wersję 2.x, jak i 3.x
Której wersji Pythona powiniem użyć?
Wymagania wstępne dla użytkowników tej książki
Struktura tej książki
Czym nie jest ta książka
 Ta książka nie jest leksykonem ani przewodnikiem po konkretnych zastosowaniach Pythona
 To nie jest krótka historia dla ludzi, którzy się spieszą
 Ta książka jest tak liniowa, jak na to pozwala Python
O przykładach zamieszczonych w książce
 Wersje Pythona
 Platformy
 Pobieranie kodów źródłowych dla tej książki
Konwencje wykorzystywane w książce
Podziękowania
 Trochę wspomnień
 Podziękowania dla Pythona
 Podziękowania osobiste
Część I Wprowadzenie
Rozdział 1.Pytania i odpowiedzi dotyczące Pythona
 Dlaczego ludzie używają Pythona?
 Jakość oprogramowania
 Wydajność programistów
 Czy Python jest językiem skryptowym?
 Jakie są wady języka Python?
 Kto dzisiaj używa Pythona?
 Co mogę zrobić za pomocą Pythona?
 Programowanie systemowe
 Graficzne interfejsy użytkownika (GUI)
 Skrypty internetowe
 Integracja komponentów
 Programowanie bazodanowe
 Szybkie prototypowanie
 Programowanie numeryczne i naukowe
 I dalej: gry, przetwarzanie obrazu, wyszukiwanie danych, robotyka, Excel...
 Jak Python jest rozwijany i wspierany?
 Kompromisy związane z modelem open source
 Jakie są techniczne mocne strony Pythona?
 Jest zorientowany obiektowo i funkcyjny
 Jest darmowy
 Jest przenośny
 Ma duże możliwości
 Mogę go łączyć z innymi językami
 Jest względnie łatwy w użyciu
 Jest względnie łatwy do nauczenia się
 Zawdzięcza swoją nazwę Monty Pythonowi
 Jak Python wygląda na tle innych języków?
 Podsumowanie rozdziału
 Sprawdź swoją wiedzę — quiz
 Sprawdź swoją wiedzę — odpowiedzi
Rozdział 2. Jak Python wykonuje programy?

[Wprowadzenie do interpretera Pythona](#)
[Wykonywanie programu](#)
 [Z punktu widzenia programisty.](#)
 [Z punktu widzenia Pythona](#)
 [Kompilacja kodu bajtowego](#)
 [Maszyna wirtualna Pythona](#)
 [Wpływ na wydajność](#)
 [Wpływ na proces programowania](#)
[Warianty modeli wykonywania](#)
 [Alternatywne implementacje Pythona](#)
 [CPython — standard](#)
 [Jython — Python dla języka Java](#)
 [IronPython — Python dla .NET](#)
 [Stackless: Python dla programowania współbieżnego](#)
 [PyPy — Python dla szybkości i wydajności](#)
 [Narzędzia do optymalizacji działania programu](#)
 [Cython: hybryda Pythona/C](#)
 [Shed Skin: translator języka Python na C++](#)
 [Psyco — oryginalny kompilator JIT](#)
 [Zamrożone pliki binarne](#)
 [Przyszłe możliwości?](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 3. Jak wykonuje się programy?](#)

[Interaktywny wiersz poleceń](#)
 [Uruchamianie sesji interaktywnej](#)
 [Ścieżka systemowa](#)
 [Nowe opcje systemu Windows w wersji 3.3: PATH, Launcher](#)
 [Gdzie zapisywać programy — katalogi z kodem źródłowym](#)
 [Czego nie wpisywać — znaki zachęty i komentarze](#)
 [Interaktywne wykonywanie kodu](#)
 [Do czego służy sesja interaktywna](#)
 [Eksperymentowanie](#)
 [Testowanie](#)
 [Uwagi praktyczne — wykorzystywanie sesji interaktywnej](#)
 [Wpisywanie instrukcji wielowierszowych](#)
 [Systemowy wiersz poleceń i pliki źródłowe](#)
 [Pierwszy skrypt](#)
 [Wykonywanie plików z poziomu wiersza poleceń powłoki](#)
 [Sposoby użycia wiersza poleceń](#)
 [Uwagi praktyczne — wykorzystywanie wierszy poleceń i plików](#)
 [Skrypty wykonywalne w stylu uniksowym — #!](#)
 [Podstawy skryptów uniksowych](#)
 [Sztuczka z wyszukiwaniem programu przy użyciu polecenia env w systemie Unix](#)
 [Python 3.3 launcher — #! w systemie Windows](#)

[Klikanie ikon plików](#)
 [Podstawowe zagadnienia związane z klikaniem ikon plików](#)
 [Kliknięcie ikony w systemie Windows](#)
 [Sztuczka z funkcją input](#)
 [Inne ograniczenia programów uruchamianych kliknięciem ikony](#)

[Importowanie i przeładowywanie modułów](#)
 [Podstawy importowania i przeładowywania modułów](#)
 [Więcej o modułach — atrybuty](#)
 [Moduły i przestrzenie nazw](#)
 [Uwagi praktyczne — instrukcje import i reload](#)

[Wykorzystywanie funkcji exec do wykonywania plików modułów](#)

[Interfejs użytkownika środowiska IDLE](#)
 [Szczegóły uruchamiania środowiska IDLE](#)
 [Podstawy środowiska IDLE](#)
 [Wybrane funkcje środowiska IDLE](#)

[Zaawansowane narzędzia środowiska IDLE](#)
[Uwagi praktyczne — korzystanie ze środowiska IDLE](#)
[Inne środowiska IDE](#)
[Inne opcje wykonywania kodu](#)
 [Osadzanie wywołań](#)
 [Zamrożone binarne pliki wykonywalne](#)
 [Uruchamianie kodu z poziomu edytora tekstu](#)
 [Jeszcze inne możliwości uruchamiania](#)
 [Przyszłe możliwości?](#)
[Jaką opcję wybrać?](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)
[Sprawdź swoją wiedzę — ćwiczenia do części pierwszej](#)

[Część II Typy i operacje](#)

[Rozdział 4. Wprowadzenie do typów obiektów Pythona](#)

[Hierarchia pojęć w Pythonie](#)
 [Dlaczego korzystamy z typów wbudowanych](#)
 [Najważniejsze typy danych w Pythonie](#)
 [Liczby](#)
 [Łańcuchy znaków](#)
 [Operacje na sekwencjach](#)
 [Niezmienność](#)
 [Metody specyficzne dla typu](#)
 [Uzyskiwanie pomocy](#)
 [Inne sposoby kodowania łańcuchów znaków](#)
 [Ciągi znaków w formacie Unicode](#)
 [Dopasowywanie wzorców](#)
 [Listy](#)
 [Operacje na typach sekwencyjnych](#)
 [Operacje specyficzne dla typu](#)
 [Sprawdzanie granic](#)
 [Zagnieżdżanie](#)
 [Listy składane](#)
 [Słowniki](#)
 [Operacje na odwzorowaniach](#)
 [Zagnieżdżanie raz jeszcze](#)
 [Brakujące klucze — testowanie za pomocą if](#)
 [Sortowanie kluczy — pętle for](#)
 [Iteracja i optymalizacja](#)
 [Krotki](#)
 [Do czego służą krotki](#)
 [Pliki](#)
 [Pliki binarne](#)
 [Pliki tekstowe Unicode](#)
 [Inne narzędzia podobne do plików](#)
 [Inne typy podstawowe](#)
 [Jak zepsuć elastyczność kodu](#)
 [Klasy definiowane przez użytkownika](#)
 [I wszystko inne](#)
 [Podsumowanie rozdziału](#)
 [Sprawdź swoją wiedzę — quiz](#)
 [Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 5. Typy liczbowe](#)

[Podstawy typów liczbowych Pythona](#)
 [Literałы liczbowe](#)
 [Wbudowane narzędzia liczbowe](#)
 [Operatory wyrażeń Pythona](#)
 [Połączone operatory stosują się do priorytetów](#)
 [Podwyrażenia grupowane są w nawiasach](#)
 [Pomieszczone typy poddawane są konwersji](#)

Wprowadzenie: przeciążanie operatorów i polimorfizm

Liczby w akcji

Zmienne i podstawowe wyrażenia

Formaty wyświetlania liczb

Porównania — zwykłe i łączone

Dzielenie — klasyczne, bez reszty i prawdziwe

Obsługa różnych wersji Pythona

Dzielenie bez reszty a odcinanie

Dlaczego odcinanie ma znaczenie?

Precyza liczb całkowitych

Liczby zespolone

Notacja szesnastkowa, ósemkowa i dwójkowa — literały i konwersje

Operacje na poziomie bitów

Inne wbudowane narzędzia numeryczne

Inne typy liczbowe

Typ Decimal (liczby dziesiętne)

Typ Decimal — zagadnienia podstawowe

Globalne ustawianie precyzji liczb dziesiętnych

Menedżer kontekstu dziesiętnego

Typ Fraction (liczby ułamkowe)

Typ Fraction — zagadnienia podstawowe

Dokładność numeryczna ułamków zwykłych i dziesiętnych

Konwersje ułamków i typy mieszane

Zbiory

Podstawy zbiorów w Pythonie 2.6 i wersjach wcześniejszych

Literały zbiorów w Pythonie 3.x i 2.7

Ograniczenia na obiekty niemutowalne i zbiory zamrożone

Zbiory składane w Pythonie 3.x i 2.7

Dlaczego zbiory?

Wartości Boolean

Rozszerzenia numeryczne

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

Sprawdź swoją wiedzę — odpowiedzi

Rozdział 6. Wprowadzenie do typów dynamicznych

Sprawa brakujących deklaracji typu

Zmienne, obiekty i referencje

Typy powiązane są z obiektami, a nie ze zmiennymi

Obiekty są uwalniane

Referencje współdzielone

Referencje współdzielone a modyfikacje w miejscu

Referencje współdzielone a równość

Typy dynamiczne są wszędzie

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

Sprawdź swoją wiedzę — odpowiedzi

Rozdział 7. Łańcuchy znaków

Co znajdziesz w tym rozdziale

Unicode — krótka historia

Łańcuchy znaków — podstawy

Literały łańcuchów znaków

Łańcuchy znaków w apostrofach i cudzysłowach są tym samym

Sekwencje ucieczki reprezentują znaki specjalne

Surowe łańcuchy znaków blokują sekwencje ucieczki

Potrójne cudzysłówki i apostrofy kodują łańcuchy znaków będące wielowierszowymi blokami

Łańcuchy znaków w akejii

Podstawowe operacje

Indeksowanie i wycinki

Rozszerzone wycinki — trzeci limit i obiekty wycinków

Narzędzia do konwersji łańcuchów znaków

Konwersje kodu znaków

Modyfikowanie łańcuchów znaków

Metody łańcuchów znaków

Składnia wywoywania metod

Metody typów znakowych

Przykłady metod łańcuchów znaków — modyfikowanie

Przykłady metod łańcuchów znaków — analiza składniowa tekstu

Inne często używane metody łańcuchów znaków

Oryginalny moduł string (usunięty w wersji 3.0)

Wyrażenia formatujące łańcuchy znaków

Formatowanie łańcuchów tekstu z użyciem wyrażeń formatujących — podstawy

Składnia zaawansowanych wyrażeń formatujących

Przykłady zaawansowanych wyrażeń formatujących

Wyrażenia formatujące oparte na słowniku

Formatowanie łańcuchów z użyciem metody format

Podstawy

Używanie kluczy, atrybutów i przesunięć

Zaawansowana składnia wywołań metody format

Przykłady zaawansowanego formatowania łańcuchów znaków z użyciem metody format

Porównanie metody format z wyrażeniami formatującymi

Dlaczego miałbyś korzystać z metody format

Dodatkowe możliwości: wbudowane funkcje czy ogólne techniki programowania

Elastyczna składnia odwołań: dodatkowa złożoność i nakładanie się funkcjonalności

Jawne odwołania do wartości: teraz opcjonalne i prawdopodobnie nie będą używane

Nazwy metod i argumenty neutralne kontekstowo — estetyka kodu kontra zastosowania praktyczne

Funkcje a wyrażenia: niewielka wygoda

Generalne kategorie typów

Typy z jednej kategorii współdzielą zbiory operacji

Typy mutowalne można modyfikować w miejscu

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

Sprawdź swoją wiedzę — odpowiedzi

Rozdział 8. Listy oraz słowniki

Listy

Listy w akcji

Podstawowe operacje na listach

Iteracje po listach i składanie list

Indeksowanie, wycinki i macierze

Modyfikacja list w miejscu

Przypisywanie do indeksu i wycinków

Wywołania metod list

Kilka słów o sortowaniu list

Inne, często stosowane metody list

Inne popularne operacje na listach

Słowniki

Słowniki w akcji

Podstawowe operacje na słownikach

Modyfikacja słowników w miejscu

Inne metody słowników

Przykład — baza danych o filmach

Przykład — mapowanie wartości na klucze

Uwagi na temat korzystania ze słowników

Wykorzystywanie słowników do symulowania elastycznych list — liczby całkowite jako klucze

Wykorzystywanie słowników z rzadkimi strukturami danych — krotki jako klucze

Unikanie błędów z brakującymi kluczami

Zagnieżdżanie słowników

Inne sposoby tworzenia słowników

Zmiany dotyczące słowników w Pythonie 3.x i 2.7

Słowniki składane w wersjach 3.x i 2.7

Widoki słowników w wersji 3.x (oraz wersji 2.7 przy użyciu nowych metod)

[Widoki słowników i zbiory](#)
[Sortowanie kluczy słowników w wersji 3.x](#)
[Porównywanie rozmiarów słowników nie działa w 3.x](#)
[W wersji 3.x metoda has_key nie istnieje, niech żyje in!](#)

[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

Rozdział 9. Krotki, pliki i wszystko inne

[Krotki](#)
 [Krotki w akcji](#)
 [Właściwości składni krotek — przecinki i nawiasy](#)
 [Konwersje, metody oraz niemutowalność](#)
 [Dlaczego istnieją listy i krotki](#)
 [Repetytorium: rekordy — krotki nazwane](#)

[Pliki](#)
 [Otwieranie plików](#)
 [Wykorzystywanie plików](#)
 [Pliki w akcji](#)
 [Pliki tekstowe i binarne — krótka historia](#)
 [Przechowywanie obiektów Pythona w plikach i przetwarzanie ich](#)
 [Przechowywanie natywnych obiektów Pythona — moduł pickle](#)
 [Przechowywanie obiektów Pythona w formacie JSON](#)
 [Przechowywanie spakowanych danych binarnych — moduł struct](#)
 [Menedżery kontekstu plików](#)
 [Inne narzędzia powiązane z plikami](#)
 [Przegląd i podsumowanie podstawowych typów obiektów](#)
 [Elastyczność obiektów](#)
 [Referencje a kopie](#)
 [Porównania, testy równości i prawda](#)
 [Porównywania i sortowania typów mieszanych w Pythonie 2.x i 3.x](#)
 [Porównywanie słowników w Pythonie 2.x i 3.x](#)
 [Prawda czy fałsz, czyli znaczenie True i False w Pythonie](#)
 [Obiekt None](#)
 [Typ bool](#)
 [Hierarchie typów Pythona](#)
 [Obiekty typów](#)
 [Inne typy w Pythonie](#)

[Pułapki typów wbudowanych](#)
 [Przypisanie tworzy referencje, nie kopie](#)
 [Powtórzenie dodaje jeden poziom zagębszenia](#)
 [Uwaga na cykliczne struktury danych](#)
 [Typów niemutowalnych nie można modyfikować w miejscu](#)

[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)
[Sprawdź swoją wiedzę — ćwiczenia do części drugiej](#)

Część III Instrukcje i składnia

Rozdział 10. Wprowadzenie do instrukcji Pythona

[Raz jeszcze o hierarchii pojęciowej języka Python](#)
[Instrukcje Pythona](#)
[Historia dwóch if](#)
 [Co dodaje Python](#)
 [Co usuwa Python](#)
 [Nawiasy są opcjonalne](#)
 [Koniec wiersza jest końcem instrukcji](#)
 [Koniec wcięcia to koniec bloku](#)
 [Skąd bierze się składnia z użyciem wcięć](#)
 [Kilka przypadków specjalnych](#)
 [Przypadki specjalne dla reguły o końcu wiersza](#)
 [Przypadki specjalne dla reguły o wcięciach bloków](#)

[Szybki przykład — interaktywne pętle](#)

[Prosta pętla interaktywna](#)
[Wykonywanie obliczeń na danych wpisywanych przez użytkownika](#)
[Obsługa błędów poprzez sprawdzanie danych wejściowych](#)
[Obsługa błędów za pomocą instrukcji try](#)
 [Obsługa liczb zmiennoprzecinkowych](#)
[Kod zagnieżdżony na trzy poziomy głębokości](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 11. Przypisania, wyrażenia i wyświetlanie](#)

[Instrukcje przypisania](#)
 [Formy instrukcji przypisania](#)
 [Przypisanie sekwencji](#)
 [Zaawansowane wzorce przypisywania sekwencji](#)
[Rozszerzona składnia rozpakowania sekwencji w Pythonie 3.x](#)
 [Rozszerzona składania rozpakowania w działaniu](#)
 [Przypadki brzegowe](#)
 [Wygodny gadżet](#)
 [Zastosowanie w pętli for](#)
 [Przypisanie z wieloma celami](#)
 [Przypisanie z wieloma celami a współdzielone referencje](#)
[Przypisania rozszerzone](#)
 [Przypisania rozszerzone a współdzielone referencje](#)
[Reguły dotyczące nazw zmiennych](#)
 [Konwencje dotyczące nazewnictwa](#)
 [Nazwy nie mają typu, ale obiekty tak](#)

[Instrukcje wyrażeń](#)
 [Instrukcje wyrażeń i modyfikacje w miejscu](#)

[Polecenia print](#)
 [Funkcja print z Pythona 3.x](#)
 [Format wywołania](#)
 [Funkcja print z wersji 3.x w działaniu](#)
 [Instrukcja print w Pythonie 2.x](#)
 [Formy instrukcji](#)
 [Instrukcja print Pythona 2.x w działaniu](#)
 [Przekierowanie strumienia wyjściowego](#)
 [Program „ Witaj, świecie! ”](#)
 [Ręczne przekierowanie strumienia wyjścia](#)
 [Automatyczne przekierowanie strumienia](#)
 [Wyświetlanie niezależne od wersji](#)
 [Konwerter 2to3](#)
 [Importowanie z future](#)
 [Neutralizacja różnic w wyświetlaniu za pomocą kodu](#)

[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 12. Testy if i reguły składni](#)

[Instrukcje if](#)
 [Ogólny format](#)
 [Proste przykłady](#)
 [Rozgałęzienia kodu](#)
 [Obsługa domyślnych wartości wyboru](#)
 [Obsługa bardziej złożonych operacji](#)

[Reguły składni Pythona raz jeszcze](#)
 [Ograniczniki bloków — reguły tworzenia wcięć](#)
 [Unikaj mieszania tabulatorów i spacji — nowa opcja sprawdzania błędów w Pythonie 3.x](#)
 [Ograniczniki instrukcji — wiersze i znaki kontynuacji](#)
 [Kilka przypadków specjalnych](#)

[Testy prawdziwości i testy logiczne](#)
[Wyrażenie trójargumentowe if/else](#)
[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 13. Pętle while i for](#)

[Pętle while](#)

[Ogólny format](#)

[Przykłady](#)

[Instrukcje break, continue, pass oraz else w pętli](#)

[Ogólny format pętli](#)

[Instrukcja pass](#)

[Instrukcja continue](#)

[Instrukcja break](#)

[Klauzula else pętli](#)

[Więcej o części pętli else](#)

[Pętle for](#)

[Ogólny format](#)

[Przykłady](#)

[Podstawowe zastosowanie](#)

[Inne typy danych](#)

[Przypisanie krotek w pętli for](#)

[Rozszerzone przypisanie sekwencji w pętlach for w Pythonie 3.x](#)

[Zagnieżdżone pętle for](#)

[Techniki tworzenia pętli](#)

[Pętle z licznikami — range](#)

[Skanowanie sekwencji — pętla while z funkcją range kontra pętla for](#)

[Przetasowania sekwencji — funkcje range i len](#)

[Przechodzenie niewyczerpujące — range kontra wycinki](#)

[Modyfikowanie list — range kontra listy składane](#)

[Przechodzenie równolegle — zip oraz map](#)

[Równoznaczność funkcji map w Pythonie 2.x](#)

[Tworzenie słowników za pomocą funkcji zip](#)

[Generowanie wartości przesunięcia i elementów — enumerate](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 14. Iteracje i listy składane](#)

[Iteracje — pierwsze spojrzenie](#)

[Protokół iteracyjny — iteratory plików](#)

[Iterowanie ręczne — iter i next](#)

[Pełny protokół iteracji](#)

[Iteracje ręczne](#)

[Inne wbudowane typy iterowalne](#)

[Listy składane — wprowadzenie](#)

[Podstawy list składanych](#)

[Wykorzystywanie list składanych w plikach](#)

[Rozszerzona składnia list składanych](#)

[Klauzula filtrująca: if](#)

[Zagnieżdżone pętle: klauzula for](#)

[Inne konteksty iteracyjne](#)

[Nowe obiekty iterowalne w Pythonie 3.x](#)

[Wpływ na kod w wersji 2.x — zalety i wady](#)

[Obiekt iterowalny range](#)

[Obiekty iterowalne map, zip i filter](#)

[Iteratory wielokrotne kontra pojedyncze](#)

[Obiekty iterowalne — widoki słownika](#)

[Inne zagadnienia związane z iteracjami](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 15. Wprowadzenie do dokumentacji](#)

[Źródła dokumentacji Pythona](#)

[Komentarze ze znakami #](#)

[Funkcja dir](#)
[Notki dokumentacyjne — doc](#)
 [Notki dokumentacyjne zdefiniowane przez użytkownika](#)
 [Standary i priorytety notek dokumentacyjnych](#)
 [Wbudowane notki dokumentacyjne](#)
[PyDoc — funkcja help](#)
[PyDoc — raporty HTML](#)
 [Python 3.2 i nowsze wersje; tryb PyDoc dla wszystkich przeglądarek](#)
 [Python 3.2 i wersje wcześniejsze; klient GUI](#)
[Nie tylko notki docstrings — pakiet Sphinx](#)
[Zbiór standardowej dokumentacji](#)
[Zasoby internetowe](#)
[Publikowane książki](#)
[Często spotykane problemy programistyczne](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)
[Sprawdź swoją wiedzę — ćwiczenia do części trzeciej](#)
[Część IV Funkcje i generatory](#)
[Rozdział 16. Podstawy funkcji](#)
 [Dlaczego używamy funkcji](#)
 [Tworzenie funkcji](#)
 [Instrukcje def](#)
 [Instrukcja def uruchamiana jest w czasie wykonania](#)
 [Pierwszy przykład — definicje i wywoływanie](#)
 [Definicja](#)
 [Wywołanie](#)
 [Polimorfizm w Pythonie](#)
 [Drugi przykład — przecinające się sekwencje](#)
 [Definicja](#)
 [Wywołania](#)
 [Raz jeszcze o polimorfizmie](#)
 [Zmienne lokalne](#)
 [Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)
[Rozdział 17. Zasięgi](#)
 [Podstawy zasięgów w Pythonie](#)
 [Reguły dotyczące zasięgów](#)
 [Rozwiązywanie nazw — reguła LEGB](#)
 [Inne zasięgi Pythona — przegląd](#)
 [Przykład zasięgu](#)
 [Zasięg wbudowany](#)
 [Przedefiniowanie wbudowanych nazw: lepiej czy gorzej?](#)
 [Instrukcja global](#)
 [Projektowanie programów: minimalizowanie stosowania zmiennych globalnych](#)
 [Projektowanie programów: minimalizowanie modyfikacji dokonywanych pomiędzy plikami](#)
 [Inne metody dostępu do zmiennych globalnych](#)
 [Zasięgi a funkcje zagnieżdżone](#)
 [Szczegóły dotyczące zasięgów zagnieżdżonych](#)
 [Przykłady zasięgów zagnieżdżonych](#)
 [Funkcje fabrykujące: domknięcie](#)
 [Proste funkcje fabrykujące](#)
 [Funkcje fabrykujące kontra klasy, runda pierwsza](#)
 [Zachowywanie stanu zasięgu zawierającego za pomocą argumentów domyślnych](#)
 [Zasięgi zagnieżdżone, wartości domyślne i wyrażenia lambda](#)
 [Zmienne pętli mogą wymagać wartości domyślnych, a nie zasięgów](#)
 [Dowolne zagnieżdżanie zasięgów](#)
 [Instrukcja nonlocal w Pythonie 3.x](#)
 [Podstawy instrukcji nonlocal](#)
 [Instrukcja nonlocal w akcji](#)

Użycie zmiennych nielokalnych w celu modyfikacji

Przypadki graniczne

Czemu służą zmienne nonlocal? Opcje zachowania stanu

Zachowanie stanu: zmienne nonlocal (tylko w wersji 3.x)

Zachowanie stanu: zmienne globalne — tylko jedna kopia

Zachowanie stanu: klasy — jawne atrybuty (wprowadzenie)

Zachowanie stanu: atrybuty funkcji (w wersjach 3.x i 2.x)

Zachowanie stanu: obiekty mutowalne — duchy przeszłości języka Python?

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

Sprawdź swoją wiedzę — odpowiedzi

Rozdział 18. Argumenty

Podstawy przekazywania argumentów

Argumenty a współdzielone referencje

Unikanie modyfikacji argumentów mutowalnych

Symulowanie parametrów wyjścia i wielu wyników działania

Specjalne tryby dopasowywania argumentów

Podstawy dopasowywania argumentów

Składnia dopasowania argumentów

Dopasowywanie argumentów — szczegóły

Przykłady ze słowami kluczowymi i wartościami domyślnymi

Słowa kluczowe

Wartości domyślne

Łączenie słów kluczowych i wartości domyślnych

Przykłady dowolnych argumentów

Nagłówki: zbieranie argumentów

Wywołania: rozpakowywanie argumentów

Ogólne zastosowanie funkcji

Zlikwidowana wbudowana funkcja apply (Python 2.x)

Argumenty tylko ze słowami kluczowymi (z Pythona 3.x)

Reguły dotyczące kolejności

Czemu służą argumenty ze słowami kluczowymi?

Przykład z funkcją obliczającą minimum

Penne rozwiązanie

Dodatkowy bonus

Puenta

Uogólnione funkcje działające na zbiorach

Emulacja funkcji print z Pythona 3.0

Wykorzystywanie argumentów ze słowami kluczowymi

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

Sprawdź swoją wiedzę — odpowiedzi

Rozdział 19. Zaawansowane zagadnienia dotyczące funkcji

Koncepcje projektowania funkcji

Funkcje rekurencyjne

Sumowanie z użyciem rekurencji

Implementacje alternatywne

Pętle a rekurencja

Obsługa dowolnych struktur

Rekurencja kontra kolejki i stosy

Cykle, ścieżki i ograniczenia stosu

Więcej przykładów rekurencji

Obiekty funkcji — atrybuty i adnotacje

Pośrednie wywołania funkcji — obiekty „pierwszej klasy”

Introspekcja funkcji

Atrybuty funkcji

Adnotacje funkcji w Pythonie 3.x

Funkcje anonimowe — lambda

Podstawy wyrażeń lambda

Po co używamy wyrażeń lambda

Wielotorowe rozgałęzienia kodu — finał

[Jak \(nie\) zaciemniać kodu napisanego w Pythonie](#)

[Zasięgi: wyrażenia lambda również można zagnieździć](#)

[Narzędzia programowania funkcyjnego](#)

[Odwzorowywanie funkcji na obiekty iterowalne — map](#)

[Wybieranie elementów obiektów iterowalnych — funkcja filter](#)

[Łączenie elementów obiektów iterowalnych — funkcja reduce](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 20. Listy składane i generatory](#)

[Listy składane i narzędzia funkcyjne](#)

[Listy składane kontra funkcja map](#)

[Dodajemy warunki i pętle zagnieżdżone — filter](#)

[Formalna składnia list składanych](#)

[Przykład — listy składane i macierze](#)

[Nie nadużywaj list składanych: reguła KISS](#)

[Druga strona medalu: wydajność, zwięzłość, ekspresyjność](#)

[Funkcje i wyrażenia generatorów](#)

[Funkcje generatorów — yield kontra return](#)

[Zawieszanie stanu](#)

[Integracja protokołu iteracji](#)

[Funkcje generatorów w działaniu](#)

[Dlaczego funkcje generatorów?](#)

[Rozszerzony protokół funkcji generatorów — send kontra next](#)

[Wyrażenia generatorów — obiekty iterowalne spotykają złożenia](#)

[Dlaczego wyrażenia generatora?](#)

[Wyrażenia generatora a funkcja map](#)

[Wyrażenia generatora a filtry](#)

[Funkcje generatorów a wyrażenia generatorów](#)

[Generatory są obiektami o jednoprzepięgowej iteracji](#)

[Generowanie wyników we wbudowanych typach, narzędziach i klasach](#)

[Generatory i narzędzia biblioteczne: skanery katalogów](#)

[Generatory i funkcje aplikacji](#)

[Przegląd: obiekty iterowalne definiowane przez użytkownika w klasach](#)

[Przykład — generowanie mieszanych sekwencji](#)

[Sekwencje mieszające](#)

[Proste funkcje](#)

[Funkcje generatora](#)

[Wyrażenia generatora](#)

[Funkcja tester](#)

[Permutacje: wszystkie możliwe kombinacje](#)

[Nie nadużywaj generatorów: reguła EIBTI](#)

[Inne spojrzenie: miejsce i czas, zwięzłość, ekspresyjność](#)

[Przykład — emulowanie funkcji zip i map za pomocą narzędzi iteracyjnych](#)

[Tworzymy własną implementację funkcji map](#)

[Własna wersja funkcji zip\(...\) i map\(None,...\)](#)

[Podsumowanie obiektów składanych](#)

[Zakresy i zmienne składane](#)

[Zrozumieć zbiory i słowniki składane](#)

[Rozszerzona składnia zbiorów i słowników składanych](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 21. Wprowadzenie do pomiarów wydajności](#)

[Pomiary wydajności iteracji](#)

[Moduł pomiaru czasu domowej roboty](#)

[Skrypt mierzący wydajność](#)

[Wyniki pomiarów czasu](#)

[Wpływ wywołań funkcji: map](#)

[Inne rozwiązania dla modułu do pomiaru czasu](#)

[Użycie argumentów ze słowami kluczowymi w wersji 3.x](#)

Inne sugestie

[Mierzenie czasu iteracji z wykorzystaniem modułu timeit](#)

[Podstawowe reguły korzystania z modułu timeit](#)

[Interaktywne użycie i wywołania API](#)

[Korzystanie z poziomu wiersza polecenia](#)

[Mierzenie czasu działania instrukcji wielowierszowych](#)

[Inne tryby użytkowania: instalacje, podsumowania i obiekty](#)

[Moduł i skrypt testujący z użyciem modułu timeit](#)

[Wyniki działania skryptu testującego](#)

[Jeszcze trochę zabawy z mierzeniem wydajności](#)

[Wygrana funkcji map i rzadka porażka PyPy](#)

[Jeszcze kilka słów o wpływie wywołań funkcji](#)

[Techniki porównywania – własne funkcje kontra moduł timeit](#)

[Możliwości ulepszenia – kod instalacyjny](#)

[Inne zagadnienia związane z mierzeniem szybkości działania kodu – test pystone](#)

[Pułapki związane z funkcjami](#)

[Lokalne nazwy są wykrywane w sposób statyczny](#)

[Wartości domyślne i obiekty mutowalne](#)

[Funkcje, które nie zwracają wyników](#)

[Różne problemy związane z funkcjami](#)

[Otaczanie zasięgów i zmennych pętli: funkcje fabrykujące](#)

[Ukrywanie wbudowanych funkcji przez przypisania: cieniowanie](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę – quiz](#)

[Sprawdź swoją wiedzę – odpowiedzi](#)

[Sprawdź swoją wiedzę – ćwiczenia do części czwartej](#)

Część V Moduły i pakiety

Rozdział 22. Moduły – wprowadzenie

[Dlaczego używamy modułów](#)

[Architektura programu w Pythonie](#)

[Struktura programu](#)

[Importowanie i atrybuty](#)

[Moduły biblioteki standardowej](#)

[Jak działa importowanie](#)

[1. Odszukanie modułu](#)

[2. Kompilowanie \(o ile jest to potrzebne\)](#)

[3. Wykonanie](#)

[Pliki kodu bajтовego – __pycache__ w Pythonie 3.2+](#)

[Modele plików kodu bajтовego w akcji](#)

[Ścieżka wyszukiwania modułów](#)

[Konfigurowanie ścieżki wyszukiwania](#)

[Wariacje ścieżki wyszukiwania modułów](#)

[Lista sys.path](#)

[Wybór pliku modułu](#)

[Kody źródłowe modułów](#)

[Priorytety wyboru](#)

[Importowanie punktów zaczepienia i plików ZIP](#)

[Pliki zoptymalizowanego kodu bajтовego](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę – quiz](#)

[Sprawdź swoją wiedzę – odpowiedzi](#)

Rozdział 23. Podstawy tworzenia modułów

[Tworzenie modułów](#)

[Nazwy modułów](#)

[Inne rodzaje modułów](#)

[Używanie modułów](#)

[Instrukcja import](#)

[Instrukcja from](#)

[Instrukcja from *](#)

[Operacja importowania jest przeprowadzana tylko raz](#)

[Kod inicjalizujący](#)

[Instrukcje import oraz from są przypisaniami](#)
[Modyfikowanie elementów mutowalnych w modułach](#)
[Modyfikowanie nazw pomiędzy plikami](#)
[Równoważność instrukcji import oraz from](#)
[Potencjalne pułapki związane z użyciem instrukcji from](#)
[Kiedy wymagane jest stosowanie instrukcji import](#)

[Przestrzenie nazw modułów](#)
[Pliki generują przestrzenie nazw](#)
[Słowniki przestrzeni nazw: dict](#)
[Kwalifikowanie nazw atrybutów](#)
[Importowanie a zasięg](#)
[Zagnieżdżanie przestrzeni nazw](#)

[Przeładowywanie modułów](#)
[Podstawy przeładowywania modułów](#)
[Przykład przeładowywania z użyciem reload](#)

[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 24. Pakiety modułów](#)
[Podstawy importowania pakietów](#)
[Pakiety a ustawienia ścieżki wyszukiwania](#)
[Pliki pakietów init .py](#)
[Role pliku inicializacji pakietu](#)

[Przykład importowania pakietu](#)
[Instrukcja from a instrukcja import w importowaniu pakietów](#)

[Do czego służy importowanie pakietów](#)
[Historia trzech systemów](#)

[Względne importowanie pakietów](#)
[Zmiany w Pythonie 3.0](#)
[Podstawy importowania względnego](#)
[Do czego służą importy względne](#)
[Importowanie względne w wersji 3.x](#)
[Względne importy a bezwzględne ścieżki pakietów](#)

[Zasięg importów względnych](#)
[Podsumowanie reguł wyszukiwania modułów](#)
[Importy względne w działaniu](#)
[Importowanie spoza pakietów](#)
[Importy wewnętrz pakietów](#)
[Importy są nadal względne w stosunku do bieżącego katalogu roboczego](#)
[Użycie importów względnych i bezwzględnych](#)
[Importy względne przeszukują tylko pakiety](#)
[Importy są nadal względne w stosunku do katalogu roboczego \(cd.\)](#)

[Pułapki związane z importem względnym w pakietach: zastosowania mieszane](#)
[Problem](#)
[Rozwiązywanie nr 1: podkatalogi pakietów](#)
[Rozwiązywanie 2: import bezwzględny z użyciem pełnej ścieżki](#)
[Przykład: aplikacja z kodem autotestu modułu \(wprowadzenie\)](#)

[Pakiety przestrzeni nazw w Pythonie 3.3](#)
[Semantyka pakietów przestrzeni nazw](#)
[Algorytm importu](#)
[Wpływ na zwykłe pakiety: opcjonalne pliki init .py](#)
[Pakiety przestrzeni nazw w akcji](#)
[Zagnieżdżanie pakietów przestrzeni nazw](#)
[Pliki nadal mają pierwszeństwo przed katalogami](#)

[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 25. Zaawansowane zagadnienia związane z modułami](#)
[Koncepty związane z projektowaniem modułów](#)
[Ukrywanie danych w modułach](#)
[Minimalizacja niebezpieczeństw użycia from * — X oraz all](#)

[Włączanie opcji z przyszłych wersji Pythona: `future`](#)
[Mieszane tryby użycia — `name` oraz `main`](#)
 [Testy jednostkowe z wykorzystaniem atrybutu `name`](#)
[Przykład — kod działający w dwóch trybach](#)
 [Symbole walut: Unicode w akcji](#)
 [Notki dokumentacyjne: dokumentacja modułu w działaniu](#)
[Modyfikacja ścieżki wyszukiwania modułów](#)
[Rozszerzenie `as` dla instrukcji `import` oraz `from`](#)
[Przykład — moduły są obiektami](#)
[Importowanie modułów z użyciem nazwy w postaci ciągu znaków](#)
 [Uruchamianie ciągów znaków zawierających kod](#)
 [Bezpośrednie wywołania: dwie opcje](#)
[Przykład — przechodnie przeładowywanie modułów](#)
 [Przeładowywanie rekurencyjne](#)
 [Testowanie przeładowań rekurencyjnych](#)
 [Rozwiązań alternatywny](#)
 [Testowanie wariantów przeładowania](#)
[Pułapki związane z modułami](#)
 [Kolizje nazw modułów: pakiety i importowanie względne w pakietach](#)
 [W kodzie najwyższego poziomu kolejność instrukcji ma znaczenie](#)
 [Instrukcja `from` kopiuje nazwy, jednak łączy już nie](#)
 [Instrukcja `from *` może zaciemnić znaczenie zmiennych](#)
 [Funkcja `reload` może nie mieć wpływu na obiekty importowane za pomocą `from`](#)
 [Funkcja `reload` i instrukcja `from` a testowanie interaktywne](#)
 [Rekurencyjne importowanie za pomocą `from` może nie działać](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)
[Sprawdź swoją wiedzę — ćwiczenia do części piątej](#)
[Część VI Klasy i programowanie zorientowane obiektowo](#)
[Rozdział 26. Programowanie zorientowane obiektowo — wprowadzenie](#)
 [Po co używa się klas](#)
 [Programowanie zorientowane obiektowo z dystansu](#)
 [Wyszukiwanie atrybutów dziedziczonych](#)
 [Klasy a instancje](#)
 [Wywołania metod klasy](#)
 [Tworzenie drzew klas](#)
 [Przeciążanie operatorów](#)
 [Programowanie zorientowane obiektowo oparte jest na ponownym wykorzystaniu kodu](#)
 [Polimorfizm i klasy](#)
 [Programowanie przez dostosowanie](#)
 [Podsumowanie rozdziału](#)
 [Sprawdź swoją wiedzę — quiz](#)
 [Sprawdź swoją wiedzę — odpowiedzi](#)
[Rozdział 27. Podstawy tworzenia klas](#)
 [Klasy generują wiele obiektów instancji](#)
 [Obiekty klas udostępniają zachowania domyślne](#)
 [Obiekty instancji są rzeczywistymi elementami](#)
 [Pierwszy przykład](#)
 [Klasy dostosowujemy do własnych potrzeb przez dziedziczenie](#)
 [Drugi przykład](#)
 [Klasy są atrybutami w modułach](#)
 [Klasy mogą przekazywać operatory Pythona](#)
 [Trzeci przykład](#)
 [Zwracamy wyniki lub nie](#)
 [Po co przeciążamy operatory](#)
 [Najprostsza klasa Pythona na świecie](#)
 [Jeszcze kilka słów o rekordach: klasy kontra słowniki](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

Rozdział 28. Bardziej realistyczny przykład

Krok 1. — tworzenie instancji

Tworzenie konstruktów

Testowanie w miarę pracy

Wykorzystywanie kodu na dwa sposoby

Krok 2. — dodawanie metod

Tworzenie kodu metod

Krok 3. — przeciążanie operatorów

Udoskonalenie sposobów wyświetlania

Krok 4. — dostosowywanie zachowania za pomocą klas podrzędnych

Tworzenie klas podrzędnych

Rozszerzanie metod — niepoprawny sposób

Rozszerzanie metod — poprawny sposób

Polimorfizm w akcji

Dziedziczenie, dostosowanie do własnych potrzeb i rozszerzenie

Programowanie zorientowane obiektowo — idea

Krok 5. — dostosowanie do własnych potrzeb także konstruktów

Programowanie zorientowane obiektowo jest prostsze, niż się wydaje

Inne sposoby łączenia klas

Krok 6. — wykorzystywanie narzędzi do introspekcji

Specjalne atrybuty klas

Universalne narzędzie do wyświetlania

Atrybuty instancji i atrybuty klas

Nazwy w klasach narzędziowych

Ostateczna postać naszych klas

Krok 7. i ostatni — przechowywanie obiektów w bazie danych

Obiekty pickle i shelve

Moduł pickle

Moduł shelve

Przechowywanie obiektów w bazie danych za pomocą shelve

Interaktywna eksploracja obiektów shelve

Uaktualnianie obiektów w pliku shelve

Przyszłe kierunki rozwoju

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

Sprawdź swoją wiedzę — odpowiedzi

Rozdział 29. Szczegóły kodowania klas

Instrukcja class

Ogólna forma

Przykład

Metody

Przykład metody

Wywoływanie konstruktów klas nadrzędnych

Inne możliwości wywoływania metod

Dziedziczenie

Tworzenie drzewa atrybutów

Specjalizacja odziedziczonych metod

Techniki interfejsów klas

Abstrakcyjne klasy nadrzędne

Abstrakcyjne klasy nadrzędne z Pythona 3.x oraz 2.6+: wprowadzenie

Przestrzenie nazw — cała historia

Proste nazwy — globalne, o ile nie są przypisane

Nazwy atrybutów — przestrzenie nazw obiektów

Zen przestrzeni nazw Pythona — przypisania klasyfikują zmienne

Klasy zagnieżdzone — jeszcze kilka słów o regule LEGB

Słowniki przestrzeni nazw — przegląd

Lączna przestrzeń nazw — przechodzenie w górę drzewa klas

Raz jeszcze o notkach dokumentacyjnych

Klasy a moduły

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

[Sprawdź swoją wiedzę — odpowiedzi](#)

Rozdział 30. Przeciążanie operatorów

Podstawy

[Konstruktory i wyrażenia — init i sub](#)

[Często spotykane metody przeciążania operatorów](#)

[Indeksowanie i wycinanie — getitem i setitem](#)

[Wycinki](#)

[Wycinanie i indeksowanie w Pythonie 2.x](#)

[Metoda index w wersji 3.x nie służy do indeksowania!](#)

[Iteracja po indeksie — getitem](#)

[Obiekty iteratorów — iter i next](#)

[Iteratory zdefiniowane przez użytkownika](#)

[Skanowanie pojedyncze i wielokrotne](#)

[Klasy i generatory](#)

[Wiele iteracji po jednym obiekcie](#)

[Klasy i wycinki](#)

[Alternatywa: metoda iter i instrukcja yield](#)

[Wielokrotne迭代 za pomocą instrukcji yield](#)

[Test przynależności — contains, iter i getitem](#)

[Dostęp do atrybutów — getattr oraz setattr](#)

[Odwoływanie do atrybutów](#)

[Przypisywanie wartości i usuwanie atrybutów](#)

[Inne narzędzia do zarządzania atrybutami](#)

[Emulowanie prywatności w atrybutach instancji](#)

[Reprezentacje łańcuchów — repr oraz str](#)

[Po co nam dwie metody wyświetlania?](#)

[Uwagi dotyczące wyświetlania](#)

[Dodawanie prawostronne i miejscowa modyfikacja: metody radd i iadd](#)

[Dodawanie prawostronne](#)

[Stosowanie metody add w radd](#)

[Eskalowanie typu klasy](#)

[Dodawanie w miejscu](#)

[Wywołania — call](#)

[Interfejsy funkcji i kod oparty na wywołaniach zwrotnych](#)

[Porównania — lt, gt i inne](#)

[Metoda cmp w 2.x](#)

[Testy logiczne — bool i len](#)

[Metody logiczne w Pythonie 2.x](#)

[Destrukcja obiektu — del](#)

[Uwagi dotyczące stosowania destruktów](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

Rozdział 31. Projektowanie z użyciem klas

[Python a programowanie zorientowane obiektowo](#)

[Polimorfizm to interfejsy, a nie sygnatury wywołań](#)

[Programowanie zorientowane obiektowo i dziedziczenie — związek „jest”](#)

[Programowanie zorientowane obiektowo i kompozycja — związki typu „ma”](#)

[Raz jeszcze procesor strumienia danych](#)

[Programowanie zorientowane obiektowo a delegacja — obiekty „opakowujące”](#)

[Pseudoprzywatne atrybuty klas](#)

[Przegląd znieskałcania nazw zmiennych](#)

[Po co używa się atrybutów pseudoprzywatnych](#)

[Metody są obiektami — z wiązaniem i bez wiązania](#)

[W wersji 3.x metody niezwiązane są funkcjami](#)

[Metody związane i inne obiekty wywoływane](#)

[Inne obiekty wywoływane](#)

[Klasy są obiektami — uniwersalne fabryki obiektów](#)

[Do czego służą fabryki](#)

[Dziedziczenie wielokrotne — klasy mieszane](#)

[Tworzenie klas mieszanych](#)

[Odczyt listy atrybutów obiektu — dict](#)
[Wydobywanie atrybutów odziedziczonych z użyciem dir\(\)](#)
[Wypisywanie atrybutów dla każdego obiektu w drzewie klas](#)
[Uruchomienie kodu wyświetlającego drzewo](#)
[Inny przykład użycia: wyświetlenie nazw zawierających znaki podkreślenia](#)
[Inny przykład użycia: uruchomienie kodu z większymi modułami](#)
[Moduł kolektora](#)
[Miejsce na udoskonalenia: algorytm MRO, sloty, interfejsy graficzne](#)

[Inne zagadnienia związane z projektowaniem](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 32. Zaawansowane zagadnienia związane z klasami](#)

[Rozszerzanie typów wbudowanych](#)

[Rozszerzanie typów za pomocą osadzania](#)
[Rozszerzanie typów za pomocą klas podrzędnych](#)

[Klasy w nowym stylu](#)

[Jak nowy jest nowy styl](#)

[Nowości w klasach w nowym stylu](#)

[Pomijanie instancji we wbudowanych operacjach przy pobieraniu atrybutów](#)
[Dlaczego zmieniło się wyszukiwanie?](#)
[Implikacje wynikające z przechwytywania atrybutów](#)
[Wymogi kodowania obiektów pośredniczących](#)
[Więcej informacji](#)

[Zmiany w modelu typów](#)

[Konsekwencje z perspektywy kontroli typów](#)

[Wszystkie obiekty dziedziczą po klasie object](#)

[Implikacje wynikające z użycia metod domyślnych](#)

[Zmiany w dziedziczeniu diamentowym](#)

[Implikacje wynikające z dziedziczenia diamentowego](#)
[Jawne rozwiązywanie konfliktów](#)
[Zakres zmian kolejności wyszukiwania](#)

[Więcej o kolejności odwzorowywania nazw](#)

[Algorytm MRO](#)
[Śledzenie algorytmu MRO](#)

[Przykład — wiązanie atrybutów ze źródłami dziedziczenia](#)

[Nowości w klasach w nowym stylu](#)

[Sloty: deklaracje atrybutów](#)

[Podstawy slotów](#)
[Sloty i słowniki przestrzeni nazw](#)
[Wiele slotów w klasach nadrzędnych](#)
[Generyczna obsługa slotów i innych „virtualnych” atrybutów](#)
[Zasady używania slotów](#)
[Przykład stosowania slotów: klasa ListTree i funkcja mapattrs](#)
[Co z szybkością slotów?](#)

[Właściwości klas: dostęp do atrybutów](#)

[Podstawy właściwości](#)

[Narzędzia atrybutów: getattribute i deskryptory](#)

[Inne zmiany i rozszerzenia klas](#)

[Metody statyczne oraz metody klas](#)

[Do czego potrzebujemy metod specjalnych](#)
[Metody statyczne w 2.x i 3.x](#)
[Alternatywy dla metod statycznych](#)
[Używanie metod statycznych i metod klas](#)
[Zliczanie instancji z użyciem metod statycznych](#)
[Zliczanie instancji z metodami klas](#)
[Zliczanie instancji dla każdej z klas z użyciem metod klas](#)

[Dekoratory i metaklasy — część 1.](#)

[Podstawowe informacje o dekoratorach funkcji](#)
[Pierwsze spojrzenie na funkcję dekoratora zdefiniowaną przez użytkownika](#)
[Pierwsze spojrzenie na dekoratory klas i metaklasy](#)

Dalsza lektura

[Wbudowana funkcja super: zmiana na lepsze czy na gorsze?](#)

[Wielka debata o funkcji super](#)

[Tradycyjny, uniwersalny i ogólny sposób wywoływania klasy nadzędnej](#)

[Podstawy i kompromisy użycia funkcji super](#)

[Stara semantyka: magiczny obiekt pośredniczący w wersji 3.x](#)

[Pułapka: beztroskie stosowanie wielokrotnego dziedziczenia](#)

[Ograniczenie: przeciążanie operatorów](#)

[Różnice w użyciu w wersji 2.x: rozbudowane wywołania](#)

[Zalety funkcji super: zmiany drzewa i kierowania metod](#)

[Zmiana klasy w trakcie działania programu a funkcja super](#)

[Kooperatywne kierowanie metod w drzewie wielokrotnego dziedziczenia](#)

[Podstawy: kooperatywne wywołanie funkcji super w akcji](#)

[Ograniczenie: wymóg zakotwiczenia łańcucha wywołań](#)

[Zakresy: model „wszystko lub nic”](#)

[Elastyczność założenia dotyczącego kolejności wywołań](#)

[Dostosowywanie: zastąpienie metody](#)

[Sprzęganie: zastosowanie w mieszaniu klas](#)

[Dostosowywanie: wymóg takich samych argumentów](#)

[Podsumowanie funkcji super](#)

[Pułapki związane z klasami](#)

[Modyfikacja atrybutów klas może mieć efekty uboczne](#)

[Modyfikowanie mutowalnych atrybutów klas również może mieć efekty uboczne](#)

[Dziedziczenie wielokrotne — kolejność ma znaczenie](#)

[Zakresy w metodach i klasach](#)

[Różne pułapki związane z klasami](#)

[Rozsądnie wybieraj miejsce przechowywania atrybutu w instancji lub w klasie](#)

[Zazwyczaj wywołuj konstruktory klasy nadzędnej](#)

[Klasy wykorzystujące delegację w 3.x — getattr i funkcje wbudowane](#)

[Przesadne opakowywanie](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Sprawdź swoją wiedzę — ćwiczenia do części szóstej](#)

[Część VII Wyjątki oraz narzędzia](#)

[Rozdział 33. Podstawy wyjątków](#)

[Po co używa się wyjątków](#)

[Role wyjątków](#)

[Wyjątki w skrócie](#)

[Domyślny program obsługi wyjątków](#)

[Przechwytywanie wyjątków](#)

[Zgłaszanie wyjątków](#)

[Wyjątki zdefiniowane przez użytkownika](#)

[Działania końcowe](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 34. Szczegółowe informacje dotyczące wyjątków](#)

[Instrukcja try/except/else](#)

[Jak działa instrukcja try](#)

[Części instrukcji try](#)

[Przechwytywanie wybranych i wszystkich wyjątków](#)

[Przechwytywanie wszystkich wyjątków: pusta instrukcja except i klasa Exception](#)

[Część try/else](#)

[Przykład — zachowanie domyślne](#)

[Przykład — przechwytywanie wbudowanych wyjątków](#)

[Instrukcja try/finally](#)

[Przykład — działania kończące kod z użyciem try/finally](#)

[Połączona instrukcja try/except/finally](#)

[Składnia połączonej instrukcji try](#)

[Łączenie finally oraz except za pomocą zagnieżdżania](#)

[Przykład połączonego try](#)

[Instrukcja raise](#)

[Zgłaszanie wyjątków](#)

[Zakresy widoczności zmiennych w instrukcjach try i except](#)

[Przekazywanie wyjątków za pomocą raise](#)

[Łańcuchy wyjątków w Pythonie 3.x — raise from](#)

[Instrukcja assert](#)

[Przykład — wyłapywanie ograniczeń \(ale nie błędów!\)](#)

[Menedżery kontekstu with/as](#)

[Podstawowe zastosowanie](#)

[Protokół zarządzania kontekstem](#)

[Kilka menedżerów kontekstu w wersjach 3.1, 2.7 i nowszych](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 35. Obiekty wyjątków](#)

[Wyjątki — powrót do przeszłości](#)

[Wyjątki oparte na łańcuchach znaków znikają](#)

[Wyjątki oparte na klasach](#)

[Tworzenie klas wyjątków](#)

[Do czego służą hierarchie wyjątków](#)

[Wbudowane klasy wyjątków](#)

[Kategorie wbudowanych wyjątków](#)

[Domyślne wyświetlanie oraz stan](#)

[Własne sposoby wyświetlania](#)

[Własne dane oraz zachowania](#)

[Udostępnianie szczegółów wyjątku](#)

[Udostępnianie metod wyjątków](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 36. Projektowanie z wykorzystaniem wyjątków](#)

[Zagnieżdżanie programów obsługi wyjątków](#)

[Przykład — zagnieżdżanie przebiegu sterowania](#)

[Przykład — zagnieżdżanie składniowe](#)

[Zastosowanie wyjątków](#)

[Wychodzenie z głęboko zagnieżdżonych pętli: instrukcja go to](#)

[Wyjątki nie zawsze są błędami](#)

[Funkcje mogą sygnalizować warunki za pomocą raise](#)

[Zamykanie plików oraz połączeń z serwerem](#)

[Debugowanie z wykorzystaniem zewnętrznych instrukcji try](#)

[Testowanie kodu wewnątrz tego samego procesu](#)

[Więcej informacji na temat funkcji sys.exc_info](#)

[Wyświetlanie błędów i śladów stosu](#)

[Wskazówki i pułapki dotyczące projektowania wyjątków](#)

[Co powinniśmy opakować w try](#)

[Jak nie przechwytywać zbyt wiele — unikanie pustych except i wyjątków](#)

[Jak nie przechwytywać zbyt mało — korzystanie z kategorii opartych na klasach](#)

[Podsumowanie podstaw języka Python](#)

[Zbiór narzędzi Pythona](#)

[Narzędzia programistyczne przeznaczone do większych projektów](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Sprawdź swoją wiedzę — ćwiczenia do części siódmej](#)

[Część VIII Zagadnienia zaawansowane](#)

[Rozdział 37. Łańcuchy znaków Unicode oraz łańcuchy bajtowe](#)

[Zmiany w łańcuchach znaków w Pythonie 3.x](#)

[Podstawy łańcuchów znaków](#)

[Kodowanie znaków](#)

[Jak Python zapisuje ciągi znaków w pamięci](#)

[Typy łańcuchów znaków Pythona](#)
[Po co są stosowane różne typy ciągów?](#)
[Pliki binarne i tekstowe](#)
[Podstawy kodowania ciągów znaków](#)
[Literały tekstowe w Pythonie 3.x](#)
[Literały Unicode w Pythonie 2.x i 3.3](#)
[Literały tekstowe w Pythonie 2.x](#)
[Konwersje typów ciągów](#)
[Kod łańcuchów znaków Unicode](#)
[Kod tekstu z zakresu ASCII](#)
[Kod tekstu spoza zakresu ASCII](#)
[Kodowanie i dekodowanie tekstu spoza zakresu ASCII](#)
[Inne techniki kodowania łańcuchów Unicode](#)
[Literały bajtowe](#)
[Konwersja kodowania](#)
[Łańcuchy znaków Unicode w Pythonie 2.x](#)
[Mieszanie typów ciągów w wersji 2.x](#)
[Deklaracje typu kodowania znaków pliku źródłowego](#)
[Wykorzystywanie obiektów bytes z Pythona 3.x](#)
[Wywołania metod](#)
[Operacje na sekwencjach](#)
[Inne sposoby tworzenia obiektów bytes](#)
[Mieszanie typów łańcuchów znaków](#)
[Obiekt bytearray w wersji 3.x \(oraz 2.6 lub nowszej\)](#)
[Typ bytearray w akcji](#)
[Podsumowanie typów ciągów znaków w Pythonie 3.x](#)
[Wykorzystywanie plików tekstowych i binarnych](#)
[Podstawy plików tekstowych](#)
[Tryby tekstowy i binarny w Pythonie 2.x i 3.x](#)
[Brak dopasowania typu i zawartości w Pythonie 3.x](#)
[Wykorzystywanie plików Unicode](#)
[Odczyt i zapis Unicode w Pythonie 3.x](#)
[Kodowanie ręczne](#)
[Kodowanie danych wyjściowych pliku](#)
[Dekodowanie danych wejściowych pliku](#)
[Dekodowanie błędnych dopasowań](#)
[Obsługa BOM w Pythonie 3.x](#)
[PomijanieznacznikaBOMwNotatniku](#)
[PomijanieznacznikaBOMwPythonie](#)
[Pliki Unicode w Pythonie 2.x](#)
[Unicode w nazwach plików i w strumieniach](#)
[Nazwy plików: znaki i bajty](#)
[Zawartość strumienia: zmienne PYTHONIOENCODING](#)
[Inne zmiany w narzędziach do przetwarzania łańcuchów znaków w Pythonie 3.x](#)
[Moduł dopasowywania wzorców re](#)
[Moduł danych binarnych struct](#)
[Moduł serializacji obiektów pickle](#)
[Narzędzia do analizy składniowej XML](#)
[Podsumowanie rozdziału](#)
[Sprawdź swoją wiedzę — quiz](#)
[Sprawdź swoją wiedzę — odpowiedzi](#)
[Rozdział 38. Zarządzane atrybuty](#)
[Po co zarządza się atrybutami](#)
[Wstawianie kodu wykonywanego w momencie dostępu do atrybutów](#)
[Właściwości](#)
[Podstawy](#)
[Pierwszy przykład](#)
[Obliczanie atrybutów](#)
[Zapisywaniem właściwości w kodzie za pomocą dekoratorów](#)
[Dekoratory setter i deleter](#)
[Deskryptory](#)

Podstawy

Argumenty metod deskryptorów

Deskryptory tylko do odczytu

Pierwszy przykład

Obliczone atrybuty

Wykorzystywanie informacji o stanie w deskryptorach

Powiązania pomiędzy właściwościami a deskryptorami

Deskryptory, sloty i nie tylko

Metody getattr oraz getattribute

Podstawy

Unikanie pętli w metodach przechwytyujących atrybuty

Pierwszy przykład

Metoda getattribute

Obliczanie atrybutów

Metoda getattribute

Porównanie metod getattr oraz getattribute

Porównanie technik zarządzania atrybutami

Przechwytywanie atrybutów wbudowanych operacji

Powrót do menedżerów opartych na delegacji

Przykład — sprawdzanie poprawności atrybutów

Wykorzystywanie właściwości do sprawdzania poprawności

Testowanie kodu

Wykorzystywanie deskryptorów do sprawdzania poprawności

Opcja 1: sprawdzanie z wykorzystaniem współdzielonego stanu deskryptora instancji

Opcja 2: sprawdzanie z wykorzystaniem indywidualnego stanu instancji

Wykorzystywanie metody getattr do sprawdzania poprawności

Wykorzystywanie metody getattribute do sprawdzania poprawności

Podsumowanie rozdziału

Sprawdź swoją wiedzę — quiz

Sprawdź swoją wiedzę — odpowiedzi

Rozdział 39. Dekoratory

Czym jest dekorator

Zarządzanie wywołaniami orazinstancjami

Zarządzanie funkcjami oraz klasami

Wykorzystywanie i definiowanie dekoratorów

Do czego służą dekoratory

Podstawy

Dekoratory funkcji

Zastosowanie

Implementacja

Obsługa dekoracji metod

Dekoratory klas

Zastosowanie

Implementacja

Obsługa większej liczby instancji

Zagnieżdżanie dekoratorów

Argumenty dekoratorów

Dekoratory zarządzają także funkcjami oraz klasami

Kod dekoratorów funkcji

Śledzenie wywołań

Możliwości w zakresie zachowania informacji o stanie

Atrybuty instancji klasy

Zakresy zawierające oraz zmienne globalne

Zakresy funkcji zawierających oraz zmienne nietokalne

Atrybuty funkcji

Uwagi na temat klas I — dekorowanie metod klas

Wykorzystywanie zagnieżdzonych funkcji do dekoracji metod

Wykorzystywanie deskryptorów do dekorowania metod

Mierzenie czasu wywołania

Dekoratory a pomiar czasu wywołania

Nieuasne pomiaru czasu

[Dodawanie argumentów dekoratora](#)

[Pomiar czasu z użyciem argumentów dekoratora](#)

[Kod dekoratorów klas](#)

[Klasy singletona](#)

[Alternatywne rozwiązania](#)

[Śledzenie interfejsów obiektów](#)

[Śledzenie interfejsów za pomocą dekoratorów klas](#)

[Stosowanie dekoratorów klas z wbudowanymi typami](#)

[Uwagi na temat klas II — zachowanie większej liczby instancji](#)

[Dekoratory a funkcje zarządzające](#)

[Do czego służą dekoratory \(raz jeszcze\)](#)

[Bezpośrednie zarządzanie funkcjami oraz klasami](#)

[Przykład — atrybuty „prywatne” i „publiczne”](#)

[Implementacja atrybutów prywatnych](#)

[Szczegóły implementacji I](#)

[Dziedziczenie a delegacja](#)

[Argumenty dekoratora](#)

[Zachowywanie stanu i zakresy funkcji zawierającej](#)

[Wykorzystanie dict oraz slots \(i innych nazw wirtualnych\)](#)

[Uogólnienie kodu pod kątem deklaracji atrybutów jako publicznych](#)

[Szczegóły implementacji II](#)

[Użycie nazw pseudoprivatnych X](#)

[Złamanie prywatności](#)

[Kompromisy związane z dekoratorem](#)

[Znane problemy](#)

[Ograniczenie: delegacja metod przeciążania operatorów kończy się niepowodzeniem w Pythonie 3.x](#)

[Sposoby redefiniowania metod przeciążających operatory w wersji 3.x](#)

[Definicja śródwierszowa](#)

[Nadrzędne klasy mieszane](#)

[Warianty kodowania: routery, deskryptory, automatyzacja](#)

[Czy metody operatorów należy weryfikować?](#)

[Alternatywy implementacyjne: wstawianie getattribute, inspekcja stosu wywołań](#)

[W Pythonie nie chodzi o kontrolę](#)

[Przykład — sprawdzanie poprawności argumentów funkcji](#)

[Cel](#)

[Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych](#)

[Uogólnienie kodu pod kątem słów kluczowych i wartości domyślnych](#)

[Szczegóły implementacji](#)

[Dalsza introspekcja](#)

[Założenia dotyczące argumentów](#)

[Algorytm dopasowywania](#)

[Znane problemy](#)

[Niepoprawne wywołania](#)

[Dowolne argumenty](#)

[Zagnieżdżone dekoratory](#)

[Argumenty dekoratora a anotacje funkcji](#)

[Inne zastosowania — sprawdzanie typów \(skoro nalegamy!\).](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 40. Metaklasy.](#)

[Tworzyć metaklasy czy tego nie robić?](#)

[Zwiększać się poziomy magii](#)

[Język pełen haczyków](#)

[Wady funkcji pomocniczych](#)

[Metaklasy a dekoratory klas — runda 1.](#)

[Model metaklasy](#)

[Klasy są instancjami obiektu type](#)

[Metaklasy są klasami podrzędnymi klasy type](#)

[Protokół instrukcji class](#)

[Deklarowanie metaklas](#)

[Deklarowanie w wersji 3.x](#)

[Deklarowanie w wersji 2.x](#)

[Kierowanie metaklas w wersjach 3.x i 2.x](#)

[Tworzenie metaklas](#)

[Prosta metaklasa](#)

[Dostosowywanie tworzenia do własnych potrzeb oraz inicjalizacja](#)

[Pozostałe sposoby tworzenia metaklas](#)

[Użycie prostych funkcji fabrycznych](#)

[Przeciążenie wywołań tworzących klasy za pomocą zwykłych klas](#)

[Przeciążenie wywołań tworzących klasy za pomocą metaklas](#)

[Instancje a dziedziczenie](#)

[Metaklasa a klasa nadzędna](#)

[Dziedziczenie: pełna historia](#)

[Algorytm dziedziczenia w Pythonie: wersja prosta](#)

[Specjalny przypadek deskryptorów](#)

[Algorytm dziedziczenia w Pythonie: wersja nieco pełniejsza](#)

[Dziedziczenie przypisów atrybutów](#)

[Specjalny przypadek wbudowanych operacji](#)

[Metody metaklas](#)

[Metody metaklasy a metody klasy](#)

[Przeciążanie operatorów w metodach metaklasy](#)

[Przykład — dodawanie metod do klas](#)

[Ręczne rozszerzanie](#)

[Rozszerzanie oparte na metaklasie](#)

[Metaklasy a dekoratory klas — runda 2.](#)

[Rozszerzenie oparte na dekoratorach](#)

[Zarządzanie instancjami zamiast klasami](#)

[Metaklasa równoważna dekoratorowi klasy?](#)

[Przykład — zastosowanie dekoratorów do metod](#)

[Ręczne śledzenie za pomocą dekoracji](#)

[Śledzenie za pomocą metaklas oraz dekoratorów](#)

[Zastosowanie dowolnego dekoratora do metod](#)

[Metaklasy a dekoratory klas — runda 3. \(i ostatnia\)](#)

[Podsumowanie rozdziału](#)

[Sprawdź swoją wiedzę — quiz](#)

[Sprawdź swoją wiedzę — odpowiedzi](#)

[Rozdział 41. Wszystko, co najlepsze](#)

[Paradoks Pythona](#)

[„Opcjonalne” cechy języka](#)

[Przeciwko niepokojącym usprawnieniom](#)

[Złożoność a siła](#)

[Prostota a elitarność](#)

[Końcowe wnioski](#)

[Dokąd dalej?](#)

[Na bis: wydrukuj swój certyfikat!](#)

[Dodatki](#)

[Dodatek A Instalacja i konfiguracja](#)

[Instalowanie interpretera Pythona](#)

[Czy Python jest już zainstalowany?](#)

[Skąd pobrać Pythona](#)

[Instalacja Pythona](#)

[Konfiguracja Pythona](#)

[Zmienne środowiskowe Pythona](#)

[Jak ustawić opcje konfiguracyjne?](#)

[Zmienne powłoki systemu Unix i Linux](#)

[Zmienne DOS \(system Windows\)](#)

[Graficzny interfejs użytkownika zmiennych środowiskowych Windows](#)

[Rejestr systemu Windows](#)

[Pliki ścieżek](#)

[Opcje wiersza poleceń Pythona](#)

[Uruchamianie skryptów z argumentami](#)
[Uruchamianie kodu podanego w argumentach i pobranego ze standardowego wejścia](#)

[Uruchamianie modułów umieszczonych w ścieżce wyszukiwania](#)

[Tryby zoptymalizowany i niebuforowany](#)

[Tryb interaktywny po wykonaniu skryptu](#)

[Argumenty wiersza poleceń w Pythonie w.x](#)

[Uruchamianie Pythona 3.3 za pomocą wiersza poleceń Windows](#)

[Uzyskanie pomocy](#)

[Dodatek B Uruchamianie Pythona 3.x w systemie Windows](#)

[Dziedzictwo systemu Unix](#)

[Dziedzictwo systemu Windows](#)

[Wprowadzenie nowego programu uruchomieniowego w systemie Windows](#)

[Podręcznik do programu uruchomieniowego](#)

[Krok 1: dyrektywa wersji w pliku](#)

[Krok 2: parametry w wierszu poleceń](#)

[Krok 3: stosowanie i zmienianie ustawień domyślnych](#)

[Pułapki nowego programu uruchomieniowego](#)

[Pułapka 1: nieroznajomy w Uniksie wiersz #! uniemożliwia uruchomienie skryptu](#)

[Wpływ zmian na przykłady użyte w książce i korekta](#)

[Pułapka 2: domyślna wersja 2.x w programie uruchomieniowym](#)

[Wpływ zmian na przykłady w książce i korekta](#)

[Pułapka 3: nowa opcja modyfikacji zmiennej PATH](#)

[Podsumowanie: ostateczny wynik dla systemu Windows](#)

[Dodatek C Zmiany w języku Python a ninicjsza książka](#)

[Najważniejsze różnice między wersjami 2.x i 3.x](#)

[Zmiany w wersji 3.x](#)

[Rozszerzenia dostępne tylko w wersji 3.x](#)

[Ogólne uwagi do zmian w wersji 3.x](#)

[Zmiany w bibliotekach i narzędziach](#)

[Zmiany w standardowej bibliotece](#)

[Zmiany w narzędziach](#)

[Migracja do wersji 3.x](#)

[Zmiany opisane w piątym wydaniu: wersje 2.7, 3.2 i 3.3](#)

[Zmiany w wersji 2.7](#)

[Zmiany w wersji 3.3](#)

[Zmiany w wersji 3.2](#)

[Zmiany opisane w czwartym wydaniu: wersje 2.6, 3.0 i 3.1](#)

[Zmiany w wersji 3.1](#)

[Zmiany w wersjach 3.0 i 2.6](#)

[Niektóre elementy języka usunięte w Pythonie 3.0](#)

[Zmiany opisane w trzecim wydaniu: wersje 2.3, 2.4 i 2.5](#)

[Wcześniejsze i późniejsze zmiany w Pythonie](#)

[Dodatek D Rozwiązywanie ćwiczeń podsumowujących poszczególne części książki](#)

[Część I Wprowadzenie](#)

[Część II Typy i operacje](#)

[Część III Instrukcja i składnia](#)

[Część IV Funkcje i generatory](#)

[Część V Moduły i pakiety](#)

[Część VI Klasы i programowanie zorientowane obiektywne](#)

[Część VII Wyjątki oraz narzędzia](#)

[O autorze](#)

[Kolofon](#)