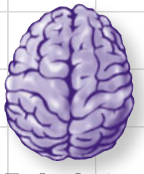
O'REILLY®

Helion

Wydanie II

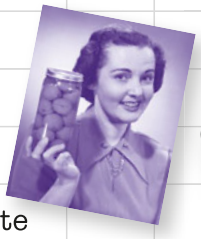
Rusz głową!

Python



Załaduj najważniejsze pojęcia Pythona wprost do swojej głowy

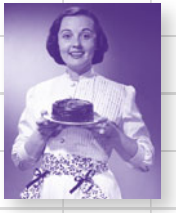
Modeluj dane jako listy, krotki, zbiory i słowniki



Nie wpadaj w opały, zamiast tego użyj DB-API Create

Obiekty? Dekoratory? Generatory? Wszystko jest na miejscu Share

Utwórz nowoczesną aplikację WWW za pomocą Flaska



Udostępniaj swój kod za pomocą modułów

Paul Barry

Tytuł oryginału: Head First Python: A Brain-Friendly Guide, 2nd Edition

Tłumaczenie: Łukasz Suma

ISBN: 978-83-283-3735-0

© 2018 Helion S.A.

Authorized Polish translation of the English edition of Head First Python, 2E ISBN 9781491919538
© 2017 Paul Barry.

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 właścicieli.

Autor oraz Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pytrg2.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pytrg2_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- Poleć książkę na Facebook.com
- Kup w wersji papierowej
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Recenzje książki *Python. Rusz głową! Wydanie II*

„Książka na temat Pythona powinna być równie zabawna jak sam ten język. Doświadczony wykładowca, Paul Barry, dostarczył nam szybkiego, zabawnego i angażującego przewodnika po tym języku w postaci książki *Python. Rusz głową!*, która świetnie przygotuje Cię do pisania prawdziwego kodu Pythona".

— Dr Eric Freeman, informatyk, dydaktyk technologiczny i były dyrektor ds. technicznych Disney Online

„Książka *Python. Rusz głową!* to doskonałe wprowadzenie zarówno do języka Python, jak i tego, jak posługiwać się nim w prawdziwym świecie. Podręcznik jest pełen praktycznych rad dotyczących pisania kodu aplikacji WWW oraz obsługi baz danych i nie unika trudnych tematów, takich jak kolekcje i niemodyfikowalność. Jeśli szukasz dobrej książki opisującej podstawy Pythona, trafiłeś na podręcznik, od którego powinieneś zacząć swoją naukę".

— David Griffiths, autor i trener metodyki Agile

„To wydanie książki *Python. Rusz głową!*, znacznie zmienione i zaktualizowane w stosunku do pierwszego wydania, z całą pewnością stanie się ulubioną pozycją w szybko powiększającym się zbiorze świetnych podręczników do nauki języka Python. Treść tej publikacji jest skonstruowana w taki sposób, że wywiera duży wpływ na czytelnika i kładzie silny nacisk na jak najszybsze rozpoczęcie praktycznego programowania. Wszystkie niezbędne zagadnienia zostały w tej książce omówione w sposób klarowny i zabawny, sprawiając, że czyta się o nich z prawdziwą przyjemnością".

— Caleb Hattingh, autor książki *20 Python Libraries You Aren't Using (But Should)* oraz kursu *Learning Cython*

„Oto czyste i gładkie wejście do basenu Pythona. Żadnych skoków na „dechę". Zanurzysz się głębiej, niż możesz się spodziewać".

— Bill Lubanovic, autor książki *Introducing Python*

Recenzje pierwszego wydania książki

Python. Rusz głową! to doskonałe wprowadzenie nie tylko do samego języka Python, lecz również tego, jak jest on używany w realnym świecie. Materiał książki wykracza poza składnię języka i uczy czytelnika, jak tworzyć aplikacje działające na telefonach z systemem Android, w środowisku App Engine firmy Google i nie tylko".

— David Griffiths, autor i trener metodyki Agile

„Podczas gdy w innych książkach zaczyna się od teorii i z czasem przechodzi się do przykładów, w przypadku *Python. Rusz głową!* wskakuje się prosto w kod i poznaje teorię wraz z postępem lektury. Sposób ten tworzy zdecydowanie bardziej efektywne środowisko do nauki, ponieważ aktywnie angażuje czytelnika od samego początku. Lektura książki to również wspaniała uciecha. Podręcznik jest zabawny, lecz nie niepoważny, zawiera też bardzo dużo wiedzy, lecz nie popada się w nim w protekcjonalny ton. Mnóstwo przykładów i wyjaśnień obejmuje większość materiału, z którym będziesz mieć do czynienia w swojej codziennej pracy. Polecam tę książkę każdemu, kto rozpoczyna przygodę z Pythonem".

— Jeremy Jones, współautor książki *Python for Unix and Linux System Administration*

Pochwały dla innych książek z serii *Rusz głową!*

„W książce *Java. Rusz głową!* Kathy i Berta drukowane strony zostały upodobnione do graficznego interfejsu użytkownika w stopniu, jakiego nigdy jeszcze nie widziałeś. W lekko drwiący, nowoczesny sposób autorzy tej książki sprawiają, że nauka Javy staje się absorbująca, a czytelnik wciąż zastanawia się: »co też oni nowego wymyślą?«".

— Warren Keuffel, *Software Development Magazine*

„Oprócz tego, że książkę *Java. Rusz głową!* cechuje zajmujący styl, który zmienia czytelnika z nowicjusza w zachwyconego wojownika Javy, opisuje ona całą masę praktycznych zagadnień, które w innych książkach są pozostawione jako przerażające »zagadnienia do opracowania we własnym zakresie«. To mądra, lekko kpiąca, nowoczesna i praktyczna pozycja — nie ma wielu książek, które mogą rościć sobie prawa do takich ocen, sprostać im, a jednocześnie uczyć czytelników serializacji obiektów i protokołów sieciowego uruchamiania aplikacji".

— Dr Dan Russell, kierownik badań nad nauką i doświadczeniami użytkowników w IBM Almaden Research Center (uczy także sztucznej inteligencji na Uniwersytecie Stanforda)

„Ta książka jest szybka, nonszalancka, zabawna i absorbująca. Uważaj — przez przypadek możesz się jeszcze czegoś nauczyć!"

— Ken Arnold, współautor (wraz z Jamesem Goslingiem, twórcą języka Java) książki *The Java Programming Language*

„Czuję się, jakby właśnie zdjęto z mojej głowy tysiące kilogramów książek".

— Ward Cunningham, wynalazca wiki i założyciel Hillside Group

„To właściwy ton dla przeciętnego, niedbale wyluzowanego mistrza koderów drzemiącego w każdym z nas. Dokonała książka do poznania praktycznych strategii programowania — zmusza mój mózg do działania bez zmuszania go do przebijania się przez wyświechtane i nieświeże profesorskie gadki".

— Travis Kalanick, współzałożyciel i prezes Ubera

„Jest wiele książek, które kupujesz, książek, które trzymasz, książek, które masz na swoim biurku, oraz — dzięki wydawnictwu O'Reilly i ekipie odpowiedzialnej za serię *Rusz głową!* — jest jeszcze jedna kategoria: książki należące do tej serii. Książki te są zwykle wygniecione, poszarpane i noszone dosłownie wszędzie. *SQL. Rusz głową!* znajduje się na samym szczycie mojego stosu. Do licha, nawet plik PDF, który otrzymałem do recenzji, jest pognieciony i poszarpany".

— Bill Sawyer, ATG Curriculum Manager w firmie Oracle

„Godna podziwu przejrzystość i humor tej książki oraz znaczna dawka wiedzy, która jest w niej zawarta, pomagają nawet osobom niebędącym programistami zacząć skutecznie myśleć o rozwiązywaniu problemów".

— Cory Doctorow, współredaktor serwisu Boing Boing, autor książek *Down and Out in the Magic Kingdom* oraz *Someone Comes to Town, Someone Leaves Town*

Pochwały dla innych książek z serii *Rusz głową!*

„Dostałem tę książkę wczoraj, zacząłem ją czytać i... nie mogłem przestać. Jest zdecydowanie très znakomita. Jest zabawna, jej autorzy opisali mnóstwo zagadnień i zawsze trafiali w punkt. Naprawdę jestem pod wrażeniem".

— Erich Gamma, IBM Distinguished Engineer oraz współautor książki
Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku

„Jedna z najzabawniejszych i najmądrzejszych książek na temat projektowania oprogramowania, jakie kiedykolwiek czytałem".

— Aaron LaBerge, wiceprezes do spraw technologii w ESPN.com

„To, co było kiedyś długim procesem nauki metodą prób i błędów, zostało obecnie zgrabnie sprowadzone do postaci interesującej książki w miękkiej okładce".

— Mike Davidson, prezes firmy Newsvine, Inc.

„Elegancki projekt to rdzeń każdego rozdziału tej książki, w której każde pojęcie zostało przedstawione z taką samą dozą pragmatyzmu, co dowcipu".

— Ken Goldstein, wiceprezes wykonawczy w Disney Online

„Kocham książkę *Head First HTML with CSS & XHTML* — uczy ona wszystkiego, co jest potrzebne, w niezwykle żartobliwy sposób".

— Sally Applin, artystka i projektantka interfejsów użytkownika

„Czytając jakąś książkę lub artykuł na temat wzorców projektowych, zwykle musiałem co jakiś czas zatrzymywać się na czymś tylko po to, aby upewnić się, że jestem odpowiednio skupiony na lekturze. Lecz nie w przypadku tej książki. Choć może to brzmieć dziwnie, książka ta sprawia, że nauka wzorców projektowych jest świetną zabawą.

Podczas gdy większość książek poświęconych wzorcom projektowym powtarza smętnie »Bueller… Bueller… Bueller…«, ta stoi na scenie, śpiewając na cały głos »Shake it up, baby!«".

— Eric Wuehler

„Dosłownie kocham tę książkę. Tak właściwie to pocałowałem ją nawet w obecności mojej żony".

— Satish Kumar

Inne książki wydawnictwa na temat Pythona:

Myśl w języku Python! Nauka programowania. Wydanie II

Python. Wprowadzenie. Wydanie IV

Python. Programuj szybko i wydajnie

Python. Leksykon kieszonkowy. Wydanie V

Python. Receptury. Wydanie III

Zaawansowany Python

TDD w praktyce. Niezawodny kod w języku Python

Inne książki z serii *Rusz głową!*

Analiza i projektowanie obiektowe. Rusz głową!

SQL. Rusz głową!

C. Rusz głową!

C#. Rusz głową! Wydanie III

Java. Rusz głową! Wydanie II

HTML5. Rusz głową!

Programowanie w JavaScript. Rusz głową!

Wzorce projektowe. Rusz głową!

jQuery. Rusz głową!

Android. Programowanie aplikacji. Rusz głową!

Mobile Web. Rusz głową!

Head First HTML with CSS & XHTML. Edycja polska

Head First JavaScript. Edycja polska

Head First Servlets & JSP. Edycja polska. Wydanie II (Rusz głową!)

Head First SQL. Edycja polska

Head Rush Ajax (Rusz głową!)

Head First Ruby on Rails. Edycja polska

Head First PHP & MySQL. Edycja polska

Head First Web Design. Edycja polska

Head First. Sieci komputerowe. Edycja polska

Pełną listę książek wydanych w ramach serii znajdziesz na stronie wydawnictwa:
http://helion.pl/serie/head-first-rusz-glowa

Niezmiennie dedykuję tę książkę wszystkim tym wspaniałomyślnym ludziom tworzącym społeczność języka Python, którzy bez ustanku pomagają sprawiać, aby był on tym, czym jest obecnie.

A także wszystkim tym osobom, które sprawiły, że Python i związane z nim technologie są na tyle skomplikowane, że ludzie potrzebują książki takiej jak ta, aby móc się ich nauczyć.

Autor książki Python. Rusz głową! Wydanie II

Podczas spaceru w lesie Paul zatrzymuje się na chwilę, aby przedyskutować prawidłową wymowę słowa „krotka" ze swoją obdarzoną anielską cierpliwością żoną.

To typowa reakcja Deirdre :)

Paul Barry mieszka i pracuje w *Irlandii*, w niewielkim miasteczku *Carlow*, które ma około 25 tys. mieszkańców i znajduje się nieco ponad 80 km na południowy zachód od stolicy kraju, *Dublina*. Paul ma tytuł licencjata z *systemów informatycznych* oraz magistra w dziedzinie *przetwarzania danych*, ukończył też studia podyplomowe z *dydaktyki*.

Paul pracował w *Instytucie Technologicznym* w *Carlow* od 1995 roku i wykładał tam od 1997 roku. Zanim zaczął uczyć, poświęcił 10 lat branży IT, pracując w Irlandii i Kanadzie, gdzie przez większość czasu zajmował się rozwiązaniami dla opieki zdrowotnej. Paul jest żonaty z Deirdre, z którą ma trójkę dzieci (dwoje z nich jest już na studiach).

Język programowania Python (oraz związane z nim technologie) stał się integralną częścią zajęć licencjackich prowadzonych przez Paula od roku akademickiego 2007.

Paul jest autorem (lub współautorem) czterech innych książek technicznych, z których dwie traktują o Pythonie, a dwie poświęcone są Perlowi. W przeszłości opracował mnóstwo materiałów dla „Linux Journal Magazine", którego był stałym współpracownikiem.

Paul wychował się w *Belfaście* w *Irlandii Północnej*, co w pewnym stopniu może tłumaczyć jego podejście do życia oraz zabawny akcent (chyba że Ty też pochodzisz z Północy; w takim razie akcent i poglądy Paula będą dla Ciebie *całkowicie normalne*).

Więcej informacji o Paulu możesz znaleźć na *Twitterze* (*@barrypj*) oraz w jego serwisie WWW: *http://paulbarry.itcarlow.ie*.

Spis treści (skrócony)

	Wprowadzenie	xxvii
1	Podstawy: *Szybki start*	1
2	Listy: *Korzystanie z uporządkowanych danych*	47
3	Dane strukturalne: *Korzystanie z danych strukturalnych*	95
4	Ponowne użycie kodu: *Funkcje i moduły*	145
5	Budowanie aplikacji WWW: *Coś praktycznego*	195
6	Przechowywanie danych i manipulowanie nimi: *Gdzie umieścić Twoje dane*	243
7	Używanie bazy danych: *Stosowanie DB-API języka Python*	281
8	Trochę klasy: *Łączenie zachowania ze stanem*	309
9	Protokół zarządzania kontekstem: *Korzystanie z instrukcji with języka Python*	335
10	Dekoratory funkcji: *Opakowywanie funkcji*	363
11	Obsługa wyjątków: *Co robić, gdy sprawy idą źle*	413
11 3/4	Trochę wątków: *Radzenie sobie z oczekiwaniem*	461
12	Zaawansowana iteracja: *Szalone zapętlenie*	477
A	Instalacja: *Instalowanie Pythona*	521
B	PythonAnywhere: *Wdrażanie Twojej aplikacji WWW*	529
C	Dziesięć najważniejszych tematów, których nie poruszyliśmy: *Zawsze można się czegoś jeszcze nauczyć*	539
D	Dziesięć najważniejszych nieomówionych projektów: *Jeszcze więcej narzędzi, bibliotek i modułów*	551
E	Zaangażowanie się: *Społeczność Pythona*	563
	Skorowidz	570

Spis treści (pełny)

Wprowadzenie

Twój mózg koncentruje się na języku Python. W tym miejscu *Ty* usiłujesz się czegoś *nauczyć*, a Twój *mózg* robi Ci przysługę i stara się, by wszystkie poznane informacje zostały *zapomniane*. Twój mózg myśli sobie: „Lepiej zostawić miejsce na naprawdę ważne informacje, takie jak dzikie zwierzęta, których należy unikać, albo to, czy jeżdżenie nago na snowboardzie jest dobrym pomysłem". W jaki sposób *możesz* przekonać swój mózg, by uznał, że Twoje życie zależy od umiejętności programowania w języku Python?

Dla kogo jest ta książka?	xxviii
Wiemy, co myślisz	xxix
Wiemy, co myśli Twój *mózg*	xxix
Metapoznanie — myślenie o myśleniu	xxxi
To, co MY zrobiliśmy	xxxii
Przeczytaj to	xxxiv
Podziękowania	xxxvii

Podstawy

1 Szybki start

Rozpocznij przygodę z Pythonem najszybciej, jak się da.

W tym rozdziale przedstawimy podstawy programowania w języku Python i zrobimy to w sposób typowy dla serii *Rusz głową!*: od razu wskakując w sam środek zagadnienia. Już po przeczytaniu kilku pierwszych stron uruchomisz swój pierwszy przykładowy program. Gdy dojdziesz do końca rozdziału, będziesz w stanie nie tylko uruchomić program, lecz również zrozumieć jego kod (i wiele więcej). W międzyczasie dowiesz się, co decyduje o charakterze **Pythona** jako języka programowania.

Znaczenie okien środowiska IDLE	4
Uruchamianie kodu instrukcja po instrukcji	8
Funkcje + moduły = biblioteka standardowa	9
Struktury danych są wbudowane	13
Wywoływanie metod daje wyniki	14
Decydowanie, kiedy należy wykonać blok kodu	15
Co jeszcze może Ci dać instrukcja warunkowa?	17
Zestawy mogą zawierać zagnieżdżone zestawy	18
Powrót do powłoki Pythona	22
Eksperymentowanie w powłoce	23
Iterowanie po sekwencji obiektów	24
Iterowanie określoną liczbę razy	25
Zastosowanie rezultatu Zadania nr 1 w naszym kodzie	26
Przygotowanie do wstrzymania wykonania	28
Generowanie losowych liczb całkowitych w Pythonie	30
Pisanie poważnej aplikacji biznesowej	38
Czy wcięcia doprowadzają Cię do szału?	40
Proszenie interpretera o pomoc dotyczącą funkcji	41
Eksperymentowanie z zakresami	42
Kod przykładów z rozdziału 1.	46

2 Listy

Korzystanie z uporządkowanych danych

Wszystkie programy przetwarzają dane, a programy Pythona nie są tu żadnym wyjątkiem.

Istotnie, rozejrzyj się tylko wokół: *dane są dosłownie wszędzie*. Bardzo znaczna część, jeśli nie wręcz większość programowania dotyczy danych: *pobierania* danych, *przetwarzania* danych, *rozumienia* danych. Aby efektywnie korzystać z danych, musisz mieć jakieś miejsce do ich *przechowywania* w czasie przetwarzania. Python błyszczy w tej kwestii na tle konkurencji dzięki (w znacznej mierze) temu, że oferuje dostęp do struktur danych o *szerokim zastosowaniu*: **list**, **słowników**, **krotek** oraz **zbiorów**. W niniejszym rozdziale przedstawimy Ci pokrótce wszystkie te cztery rodzaje struktur, a następnie poświęcimy większą część jego treści na zagłębienie się w tematykę **list** (pozostałe trzy zostaną dokładniej omówione w kolejnym rozdziale). Postanowiliśmy zaprezentować Ci struktury danych na stosunkowo wczesnym etapie, ponieważ większość rzeczy, które będziesz robił z Pythonem, będzie się kręcić wokół korzystania z danych.

Liczby, łańcuchy znakowe... i obiekty	48
Poznaj cztery wbudowane struktury danych	50
Nieuporządkowana struktura danych: słownik	52
Struktura danych, która eliminuje duplikaty: zbiór	53
Literalne tworzenie list	55
Korzystaj z edytora, gdy pracujesz nad dłuższym fragmentem kodu	57
„Rozrost” listy w czasie wykonywania	58
Sprawdzanie przynależności za pomocą operatora in	59
Usuwanie obiektów z listy	62
Poszerzanie listy o nowe obiekty	64
Wstawianie obiektu na listę	65
Jak skopiować strukturę danych	73
Listy rozszerzają notację wykorzystującą nawiasy kwadratowe	75
Listy „rozumieją” początek, koniec i krok	76
Działania na początku i końcu listy	78
Zaprzęganie wycinków do pracy z listami	80
Pętla for w języku Python „rozumie” listy	86
Wycinki listy Marvina z bliska	88
Kiedy nie używać list	91
Kod przykładów z rozdziału 2. (1. z 2)	92

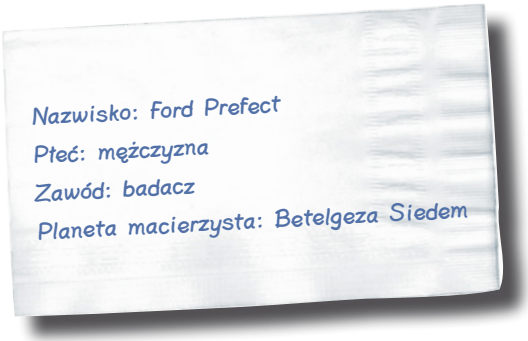
Dane strukturalne

3 Korzystanie z danych strukturalnych

Zapewniana przez język Python struktura danych, jaką jest lista, sprawdza się świetnie, ale nie stanowi remedium na wszelkie potrzeby związane z danymi.

Gdy masz do czynienia z *prawdziwie* strukturalnymi danymi (a używanie listy do ich przechowywania może nie być najlepszą opcją), Python oferuje Ci ratunek w postaci wbudowanego **słownika**. Dostępny od ręki słownik umożliwia przechowywanie wszelkich kolekcji składających się z *par klucz-wartość* i manipulowanie nimi. W tym rozdziale będziemy mu się przyglądać długo i uważnie, a gdzieś po drodze spotkamy też **zbiór** i **krotkę**. Wraz z **listą** (którą zaprezentowaliśmy w poprzednim rozdziale) struktury danych, jakimi są słownik, zbiór i krotka, zapewniają zestaw wbudowanych narzędzi, które sprawiają, że Python i dane tworzą tak potężną kombinację.

Słownik przechowuje pary klucz-wartość	96
Jak znaleźć słownik w kodzie	98
Kolejność wprowadzania NIE jest zachowywana	99
Wyszukiwanie wartości przy użyciu nawiasów kwadratowych	100
Korzystanie ze słowników w czasie wykonywania programu	101
Aktualizacja licznika częstotliwości	105
Iterowanie po słowniku	107
Iterowanie po kluczach i wartościach	108
Iterowanie po słowniku przy użyciu metody items	110
Jak dynamiczne są słowniki?	114
Unikanie wyjątków KeyError w czasie wykonania	116
Sprawdzanie przynależności za pomocą operatora in	117
Zapewnianie inicjalizacji przed użyciem	118
Zastępowanie operatora in operatorami not in	119
Zaprzęganie metody setdefault do pracy	120
Wydajne tworzenie zbiorów	124
Korzystanie z metod zbioru	125
Tworzenie przypadku dla krotek	132
Łączenie wbudowanych struktur danych	135
Uzyskiwanie dostępu do złożonych struktur danych	141
Kod przykładów z rozdziału 3. (1. z 2)	143



Ponowne użycie kodu

4 Funkcje i moduły

Ponowne wykorzystanie kodu ma kluczowe znaczenie dla budowy prostego w utrzymaniu systemu.

A jeśli chodzi o wielokrotne używanie kodu w Pythonie, wszystko zaczyna się od skromnej **funkcji** i na niej też kończy. Weź kilka wierszy kodu, nadaj im nazwę i masz funkcję (która może być wielokrotnie wykorzystana). Weź kolekcję funkcji i spakuj je do jednego pliku, a dostaniesz **moduł** (którego również można używać wiele razy). Prawdą jest, że *dobrze się dzielić*. Zanim skończysz czytać ten rozdział, będziesz na dobrej drodze do **udostępniania** i **ponownego wykorzystywania** swojego kodu, a to dzięki poznaniu sposobu działania funkcji i modułów w języku Python.

Ponowne wykorzystywanie kodu za pomocą funkcji	146
Wprowadzenie do funkcji	147
Wywoływanie Twojej funkcji	150
Funkcje przyjmują argumenty	154
Zwracanie jednej wartości	158
Zwracanie więcej niż jednej wartości	159
Przypomnienie wbudowanych struktur danych	161
Tworzenie funkcji ogólnego przeznaczenia	165
Tworzenie kolejnej funkcji (1. z 3)	166
Określanie domyślnych wartości argumentów	170
Układ pozycyjny kontra słowa kluczowe	171
Odświeżanie tego, co już wiemy o funkcjach	172
Uruchamianie Pythona z wiersza poleceń	175
Tworzenie wymaganych plików konfiguracji	179
Tworzenie pliku dystrybucji	180
Instalowanie pakietów za pomocą narzędzia pip	182
Prezentacja semantyki „wywołania przez wartość”	185
Prezentacja semantyki „wywołania przez referencję”	186
Instalacja narzędzi programistycznych do testowania	190
Jak bardzo zgodny z PEP 8 jest nasz kod?	191
Znaczenie komunikatów błędów	192
Kod przykładów z rozdziału 4.	194

moduł

5 Budowanie aplikacji WWW

Coś praktycznego

Na tym etapie znasz już Pythona wystarczająco dobrze, aby stanowić zagrożenie.

Po lekturze czterech pierwszych rozdziałów tej książki jesteś już w stanie produktywnie korzystać z Pythona w wielu różnych obszarach zastosowań (nawet pomimo tego, że nadal masz sporo do nauki). Zamiast wymieniać te liczne obszary zastosowań, w tym rozdziale i kilku kolejnych mamy zamiar skonstruować nasz wykład wokół tworzenia aplikacji WWW, a więc na polu, na którym język Python sprawdza się szczególnie dobrze. Doskonaląc swoje praktyczne umiejętności programowania, dowiesz się trochę więcej o Pythonie.

Python: co już wiemy 196
Czego oczekujemy od naszej aplikacji WWW? 200
Zainstalujmy framework Flask 202
Jak działa framework Flask? 203
Pierwsze uruchomienie aplikacji WWW wykorzystującej framework Flask 204
Tworzenie obiektu aplikacji WWW frameworka Flask 206
Dekorowanie funkcji za pomocą ciągu URL 207
Uruchamianie działań Twojej aplikacji WWW 208
Udostępnianie funkcji w sieci 209
Budowanie formularza HTML 213
Szablony mają związek ze stronami WWW 216
Renderowanie szablonów z poziomu Flaska 217
Wyświetlanie formularza HTML aplikacji WWW 218
Przygotowanie do uruchomienia kodu szablonu 219
Znaczenie kodów stanu HTTP 222
Obsługa danych przekazywanych za pomocą metody POST 223
Ulepszenie cyklu edycji, zatrzymywania, uruchamiania i testowania 224
Dostęp do danych formularza HTML za pomocą Flaska 226
Używanie danych żądania w swojej aplikacji WWW 227
Generowanie wyników w postaci kodu HTML 229
Przygotowywanie aplikacji WWW do umieszczenia w chmurze 238
Kod przykładów z rozdziału 5. 241

6 Przechowywanie danych i manipulowanie nimi

Gdzie umieścić Twoje dane

Wcześniej czy później będziesz musiał gdzieś bezpiecznie zachować swoje dane.

A gdy idzie o **przechowywanie danych**, język Python daje Ci wszystko, czego trzeba. W tym rozdziale poznasz sposoby zapisywania danych w *plikach tekstowych* i odczytywania ich z tych plików. Pliki tekstowe mogą się wydawać dość prostym rozwiązaniem, gdy weźmie się pod uwagę rozwój mechanizmów do przechowywania danych, nadal są jednak z powodzeniem stosowane w wielu różnych sytuacjach. Oprócz zapisywania danych w plikach i ich odczytywania przybliżymy Ci również pewne sztuczki związane z manipulowaniem danymi. Z prezentacją „poważnego materiału" (czyli informacji na temat przechowywania danych w bazie danych) poczekamy do następnego rozdziału, ale do tej pory mamy jeszcze do omówienia tak dużo tematów związanych z korzystaniem z plików, że na pewno nie będziemy się nudzić.

Robienie czegoś z danymi Twojej aplikacji WWW 244
Python obsługuje otwieranie, przetwarzanie i zamykanie 245
Odczytywanie danych z istniejącego pliku 246
Lepsze otwieranie, przetwarzanie i zamykanie za pomocą instrukcji with 248
Przeglądanie logu za pomocą naszej aplikacji WWW 254
Sprawdzanie surowych danych za pomocą przeglądarki źródeł strony 256
Czas na użycie sekwencji ucieczki (w stosunku do Twoich danych) 257
Przeglądanie pełnego logu w Twojej aplikacji WWW 258
Logowanie wybranych atrybutów żądania sieciowego 261
Loguj pojedynczy wiersz rozdzielonych danych 262
Od surowych danych do czytelnych danych wyjściowych 265
Generowanie czytelnych danych wyjściowych przy użyciu kodu HTML 274
Osadzanie logiki wyświetlania w swoim szablonie 275
Produkcja czytelnych danych wyjściowych przy użyciu silnika Jinja2 276
Bieżący stan kodu naszej aplikacji WWW 278
Zadawanie pytań na temat swoich danych 279
Kod przykładów z rozdziału 6. 280

Dane z formularza	Adres klienta	Agent użytkownika	Wyniki
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])|	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'i', 'e'}

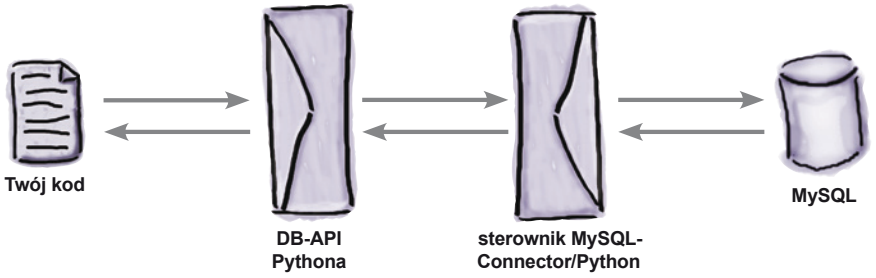
7 Używanie bazy danych

Stosowanie DB-API języka Python

Przechowywanie danych w relacyjnej bazie danych jest wygodne.

W tym rozdziale dowiesz się, jak pisać kod umożliwiający korzystanie z popularnej bazy danych **MySQL** za pomocą ogólnego interfejsu o nazwie **DB-API**. API to (zapewniane standardowo przez każdą instalację języka Python) umożliwia tworzenie kodu, który można łatwo przenosić z jednego systemu bazy danych do innych, jeśli tylko systemy te wykorzystują język SQL. Chociaż będziemy stosować MySQL-a, nic nie stoi na przeszkodzie, abyś kod używający DB-API wykorzystywał do działań na innych relacyjnych bazach danych, niezależnie od tego, jaki konkretnie system wybierzesz. Sprawdźmy zatem, jak w języku Python wygląda obsługa relacyjnej bazy danych. W rozdziale tym nie znajdziesz wielu nowych informacji na temat samego Pythona, ale używanie tego języka do komunikacji z bazami danych jest **niezwykle istotną kwestią**, dlatego warto dobrze ją poznać.

Włączanie obsługi bazy danych w Twojej aplikacji WWW 282
Zadanie nr 1: Zainstaluj serwer bazy danych MySQL 283
Wprowadzenie do DB-API języka Python 284
Zadanie nr 2: Zainstaluj sterownik bazy danych MySQL dla języka Python 285
Zainstaluj sterownik MySQL-Connector/Python 286
Zadanie nr 3: Utwórz bazę danych i tabele dla naszej aplikacji WWW 287
Wybierz strukturę dla swojego logu 288
Potwierdź, że Twoja tabela jest gotowa na przyjęcie danych 289
Zadanie nr 4: Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel 296
Przechowywanie danych to dopiero połowa sukcesu 300
Jak najlepiej ponownie użyć Twojego kodu obsługi bazy danych? 301
Zastanów się, co próbujesz ponownie wykorzystać 302
A co z tą instrukcją importowania? 303
Miałeś już wcześniej do czynienia z tym wzorcem 305
Zła wiadomość nie jest wcale aż taka zła 306
Kod przykładów z rozdziału 7. 307



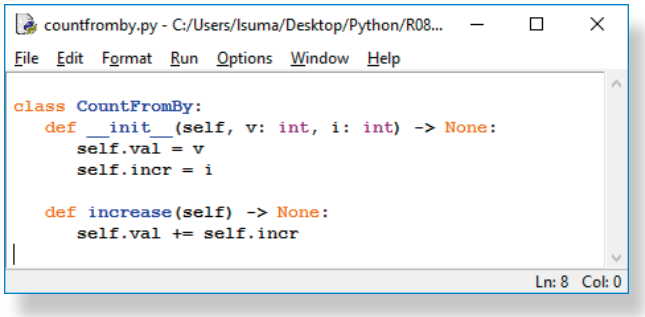
8 Trochę klasy

Łączenie zachowania ze stanem

Klasy umożliwiają łączenie kodu odpowiedzialnego za stan i działanie.

W tym rozdziale na jakiś czas zostawisz swoją aplikację WWW w spokoju i zajmiesz się nauką tworzenia **klas** w języku Python. Zrobisz to, aby uzyskać możliwość utworzenia menedżera kontekstu, w czym pomoże Ci właśnie klasa Pythona. Rozdział ten poświęcamy w całości kwestiom związanym z tworzeniem i używaniem klas, ponieważ są to rzeczy, które i tak zdecydowanie warto znać. Nie omówimy tu wszystkich tematów dotyczących klas, ale poruszymy wszystkie zagadnienia, jakie musisz zrozumieć, aby bez kłopotu utworzyć menedżer kontekstu, którego oczekuje Twoja aplikacja.

Zastosowanie instrukcji with 310
Elementarz programowania zorientowanego obiektowo 311
Tworzenie obiektów klas 312
Obiekty współdzielą zachowanie, ale nie stan 313
Więcej działań z klasą CountFromBy 314
Wywołanie metody: zrozumienie szczegółów 316
Dodawanie metody do klasy 318
Duże znaczenie self 320
Radzenie sobie z zasięgiem 321
Poprzedzaj swoje atrybuty nazwą self 322
Inicjalizuj wartości (atrybutów) przed użyciem 323
Dunder init umożliwia inicjalizowanie atrybutów 324
Inicjalizowanie atrybutów za pomocą dundera init 325
Znaczenie reprezentacji klasy CountFromBy 328
Definiowanie reprezentacji klasy CountFromBy 329
Zapewnianie sensownych wartości domyślnych dla klasy CountFromBy 330
Klasy: co już wiemy 332
Kod przykładów z rozdziału 8. 333



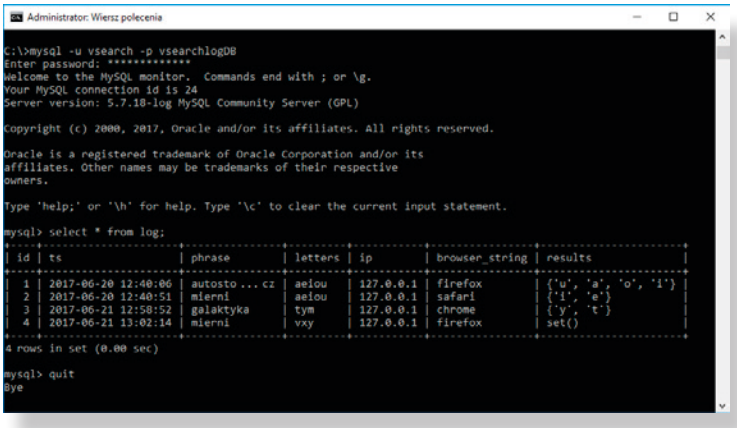
9 Protokół zarządzania kontekstem

Korzystanie z instrukcji with języka Python

Pora w praktyce skorzystać z tego, czego się właśnie nauczyłeś.

W rozdziale 7. omówiliśmy używanie **relacyjnej bazy danych** w Pythonie, natomiast rozdział 8. stanowił wstęp do korzystania z **klas** w Twoim kodzie napisanym w tym języku. W niniejszym rozdziale obydwie te rzeczy zostaną połączone w celu utworzenia **menedżera kontekstu**, który pozwoli nam rozszerzyć instrukcję `with` w taki sposób, aby mogła współpracować z systemami relacyjnych baz danych. Będziesz tu korzystać z instrukcji `with`, tworząc nową klasę, która spełnia wymagania **protokołu zarządzania kontekstem** języka Python.

W jaki sposób najlepiej współdzielić kod obsługi bazy danych naszej aplikacji WWW? 336
Zarządzanie kontekstem za pomocą metod 338
Menedżer kontekstu widziałeś już w akcji 339
Utwórz nową klasę menedżera kontekstu 340
Zainicjalizuj klasę przy użyciu konfiguracji bazy danych 341
Dokonaj konfiguracji za pomocą metody __enter__ 343
Przeprowadź demontaż za pomocą metody __exit__ 345
Zrewiduj kod swojej aplikacji WWW (1. z 2) 348
Przypomnienie funkcji log_request 350
Modyfikacja funkcji log_request 351
Przypomnienie funkcji view_the_log 352
Nie tylko kod się tu zmienia 353
Modyfikacja funkcji view_the_log 354
Odpowiadanie na pytania o dane 359
Kod przykładów z rozdziału 9. (1. z 2) 360



10

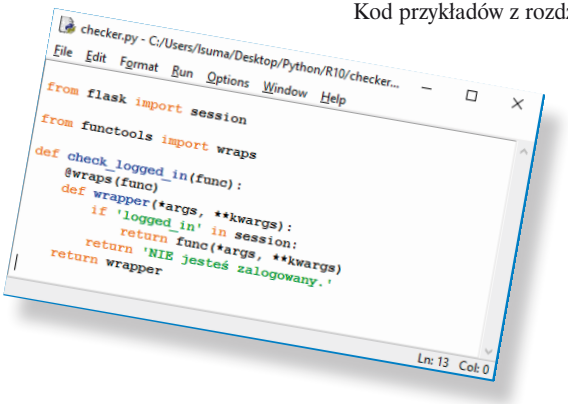
Dekoratory funkcji

Opakowywanie funkcji

Jeśli chodzi o zwiększanie możliwości Twojego kodu, przedstawiony w rozdziale 9. protokół zarządzania kontekstem nie jest jedyną dostępną opcją.

Język Python umożliwia Ci również korzystanie z **dekoratorów** funkcji, czyli techniki, dzięki której możesz dodawać kod do istniejących funkcji *bez* konieczności modyfikacji jakiegokolwiek kodu należącego do nich. Jeśli sądzisz, że masz tu do czynienia z jakimś rodzajem czarnej magii, nie załamuj się, ponieważ to nic z tych rzeczy. Mimo to wśród różnych technik programowania tworzenie dekoratorów funkcji przez wielu programistów Pythona jest nierzadko uważane za trudne rozwiązanie i z tego też powodu nie jest wykorzystywane tak często, jak powinno być. W tym rozdziale mamy zamiar pokazać Ci, że choć stanowi to zaawansowaną technikę, tworzenie i używanie własnych dekoratorów wcale nie jest aż tak trudne, jak może się niektórym wydawać.

Twój kod działa na serwerze WWW (nie na Twoim komputerze)	366
Technologia sesji frameworka Flask zapewnia stan	368
Przeszukiwanie słownika umożliwia pobranie stanu	369
Zarządzanie logowaniem za pomocą sesji	374
Zajmijmy się wylogowywaniem i sprawdzaniem stanu	377
Przekaż funkcję do funkcji	386
Wywoływanie przekazanej funkcji	387
Przyjmowanie listy argumentów	390
Przetwarzanie listy argumentów	391
Przyjmowanie słowników argumentów	392
Przetwarzanie słownika argumentów	393
Przyjmowanie dowolnej liczby i typu argumentów funkcji	394
Tworzenie dekoratora funkcji	397
Ostatni krok: obsługa argumentów	401
Zaprzęganie Twojego dekoratora do pracy	404
Powrót do ograniczania dostępu do strony /viewlog	408
Kod przykładów z rozdziału 10. (1. z 2)	410



11

Obsługa wyjątków

Co robić, gdy sprawy idą źle

Sprawy idą źle, dzieje się tak zawsze — niezależnie od tego, jak dobry jest Twój kod.

Jak do tej pory wszystkie przykłady przedstawione w tej książce bez problemu udawało Ci się uruchomić i raczej jesteś pewien, że zaprezentowany dotąd kod działa. Czy oznacza to jednak, że kod ten jest solidny? Raczej nie. Pisanie kodu w oparciu o założenie, że nic złego nie może się zdarzyć, jest (w najlepszym przypadku) oznaką naiwności. W najgorszym przypadku jest to wręcz praktyka niebezpieczna, ponieważ nieprzewidziane rzeczy się zdarzają (i będą się zdarzać). Lepiej, abyś tworząc kod, był ostrożny niż beztroski. Ostrożność jest konieczna, aby mieć pewność, że Twój kod robi to, czego od niego oczekujesz, jak również że reaguje prawidłowo, gdy rzeczy idą w złą stronę.

Bazy danych nie są zawsze dostępne 418
Ataki sieciowe to prawdziwy problem 419
Operacje wejścia-wyjścia są (czasami) powolne 420
Wywołania Twoich funkcji mogą się nie powieść 421
Kod podatny na błędy powinieneś zawsze próbować wykonywać 423
Korzystaj z jednej instrukcji try, ale wielu klauzul except 426
Kod obsługi wszystkich wyjątków 428
Poznawanie wyjątków za pomocą modułu sys 430
Zrewidowany kod obsługi wszystkich wyjątków 431
Powrót do kodu naszej aplikacji WWW 433
Cicha obsługa wyjątków 434
Obsługa innych błędów bazy danych 440
Unikaj ściśle sprzężonego kodu 442
Zrewidowany moduł DBcm 443
Tworzenie własnego wyjątku 444
Co jeszcze może pójść źle z modułem DBcm? 448
Obsługa wyjątku SQLError odbywa się w inny sposób 451
Zgłaszanie wyjątku SQLError 453
Szybkie przypomnienie: poprawianie odporności 455
Jak radzić sobie z czekaniem? To zależy... 456
Kod przykładów z rozdziału 11. (1. z 3) 457


```
...
+-- Exception
     +-- StopIteration
     +-- StopAsyncIteration
     +-- ArithmeticError
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
...
```

11 3/4

Trochę wątków

Radzenie sobie z oczekiwaniem

Twój kod może nieraz potrzebować sporo czasu, aby się wykonać.

Zależnie od tego, kto dokonuje obserwacji, to może, lecz nie musi być problemem. Jeśli wykonanie jakiegoś kodu wymaga 30 sekund, lecz odbywa się „w tle", nawet tak długie oczekiwanie wcale nie musi być kłopotliwe. Jeśli jednak użytkownik aplikacji czeka na jej odpowiedź pół minuty, nie ma takiej osoby, która mogłaby tego nie zauważyć. Sposób, w jaki powinieneś naprawić tego rodzaju problem, zależy od tego, co próbujesz robić (i kto musi czekać). W tym krótkim rozdziale pokrótce opiszemy niektóre możliwości, a następnie przyjrzymy się jednemu rozwiązaniu problemu, z którym mamy do czynienia, czyli odpowiemy na pytanie: *co się stanie, gdy coś zabiera dużo czasu?*

Oczekiwanie: co robić?	462
Jak odpytujesz swoją bazę danych?	463
Instrukcje INSERT i SELECT są różne	464
Robienie więcej niż jednej rzeczy naraz	465
Nie daj się zdołować: użyj wątków	466
Najpierw najważniejsze: nie panikuj	470
Nie daj się zdołować: pomóc może Ci framework Flask	471
Czy Twoja aplikacja jest teraz odporna?	474
Kod przykładów z rozdziału 11 3/4 (1. z 2)	475

12

Zaawansowana iteracja

Szalone zapętlenie

To zdumiewające, ile czasu nasze programy spędzają w pętlach.

Nie jest to duża niespodzianka, ponieważ zadanie większości programów polega na szybkim wykonywaniu pewnych czynności bardzo wiele razy. Jeśli chodzi o optymalizację pętli, do dyspozycji mamy dwa podstawowe sposoby: pierwszy polega na poprawieniu składni pętli (aby ułatwić definiowanie pętli), a drugi sprowadza się do poprawienia sposobu działania pętli (aby były one wykonywane szybciej). Na wczesnym etapie istnienia języka Python 2 (czyli *dawno*, *dawno* temu) jego projektanci wprowadzili pewną cechę języka, która stanowiła implementację obydwu tych sposobów i otrzymała dość dziwną nazwę: **komprehencja** (ang. *comprehension*).

Odczytywanie danych CSV jako list	479
Odczytywanie danych CSV jako słowników	480
„Obieranie”, a następnie podział Twoich surowych danych	482
Zachowaj ostrożność, łącząc wywołania metod w łańcuch	483
Przekształcanie danych na format, którego potrzebujesz	484
Przekształcanie do postaci słownika list	485
Dostrzeganie wzorców w przypadku list	490
Przekształcanie wzorców w komprehencje	491
Przyjrzyj się uważnie swojej komprehencji	492
Definiowanie komprehencji słownika	494
Rozszerz komprehencje za pomocą filtrów	495
Poradź sobie z komplikacjami w sposób charakterystyczny dla Pythona	499
Komprehencja zbioru w akcji	505
Co z „komprehencjami krotek”?	507
Nawiasy okrągłe wokół kodu = generator	508
Używanie komprehencji listy do przetwarzania URL-i	509
Używanie generatora do przetwarzania URL-i	510
Zdefiniuj, co powinna robić Twoja funkcja	512
Ulegnij potędze funkcji generatorowych	513
Śledzenie Twojej funkcji generatorowej (1. z 2)	514
Ostatnie pytanie	518
Kod przykładów z rozdziału 12.	519
Czas w drogę...	520

A

Instalacja

Instalowanie Pythona

Najpierw rzeczy najważniejsze: zainstalujmy Pythona na Twoim komputerze.

Pythona możesz swobodnie używać niezależnie od tego, czy korzystasz z systemu operacyjnego *Windows*, *Mac OS X* czy *Linux*. Sposób instalacji na tych platformach różni się jednak nieco w związku z odmiennym działaniem poszczególnych systemów (wiemy, ta wiadomość jest dla Ciebie nie lada szokiem, prawda?), lecz społeczność Pythona ciężko pracuje, aby zapewnić programy instalacyjne współpracujące ze wszystkimi popularnymi środowiskami operacyjnymi. W tym krótkim dodatku przeprowadzimy Cię przez procedurę instalacji Pythona na Twoim komputerze.

Zainstaluj Pythona 3 w systemie Windows 522
Sprawdź Pythona 3 w systemie Windows 523
Dodawanie modułów do Pythona 3 w systemie Windows 524
Zainstaluj Pythona 3 w systemie Mac OS X (macOS) 525
Sprawdź i skonfiguruj Pythona 3 w systemie Mac OS X 526
Zainstaluj Pythona 3 w systemie Linux 527

B

PythonAnywhere

Wdrażanie Twojej aplikacji WWW

Na końcu rozdziału 5. stwierdziliśmy, że na wdrożenie Twojej aplikacji WWW w chmurze potrzebnych nam było zaledwie 10 minut.

Nadszedł czas na spełnienie tej obietnicy. W tym dodatku zamierzamy przeprowadzić Cię przez procedurę wdrażania Twojej aplikacji WWW w chmurze *PythonAnywhere*, zaczynając od zera, a kończąc na pełnym wdrożeniu w około 10 minut. *PythonAnywhere* to najbardziej popularna usługa w środowisku programistów Pythona i trudno nie zauważyć, dlaczego tak właśnie jest. Rozwiązanie to działa zgodnie z oczekiwaniami, oferuje doskonałe wsparcie dla języka Python (i frameworka Flask) oraz — co najlepsze — możesz zacząć udostępniać za jego pomocą swoje aplikacje zupełnie za darmo.

Krok 0. Niewielkie przygotowania 530
Krok 1. Zarejestruj się w serwisie PythonAnywhere 531
Krok 2. Wgraj swoje pliki do chmury 532
Krok 3. Rozpakuj i zainstaluj swój kod 533
Krok 4. Utwórz starter aplikacji WWW (1. z 2) 534
Krok 5. Skonfiguruj swoją aplikację WWW 536
Krok 6. Weź w obroty swoją aplikację WWW działającą w chmurze! 537

C Dziesięć najważniejszych tematów, których nie poruszyliśmy

Zawsze można się czegoś jeszcze nauczyć

Nigdy nie było naszą intencją opisanie wszystkich tematów.

Celem tej książki było od samego początku przedstawienie takich informacji na temat Pythona, które pozwolą Ci jak najszybciej zacząć praktycznie posługiwać się tym językiem. Istnieje znacznie więcej zagadnień, które moglibyśmy tu poruszyć, ale tego nie zrobiliśmy. W tym dodatku omawiamy 10 najważniejszych rzeczy, którymi — gdybyśmy mieli kolejnych 600 (lub coś koło tego) stron do dyspozycji — zajęlibyśmy się również. Nie wszystkie z tych tematów Cię zainteresują, mimo to szybko je przejrzyj na wypadek, gdybyśmy przypadkiem trafili w jakiś Twój czuły punkt lub odpowiedzieli na dręczące Cię pytanie. Wszystkie opisane w tym dodatku rozwiązania programistyczne są z definicji dostępne w języku Python i jego interpreterze.

1. Co z Pythonem 2? 540
2. Wirtualne środowiska programistyczne 541
3. Więcej na temat obiektowości 542
4. Specyfikatory formatów łańcuchów znakowych i tym podobne 543
5. Sortowanie rzeczy 544
6. Więcej z biblioteki standardowej 545
7. Równoległe wykonywanie Twojego kodu 546
8. Graficzne interfejsy użytkownika z biblioteką tkinter (i zabawa z żółwiami) 547
9. Kod nie jest skończony, dopóki nie jest przetestowany 548
10. Debuguj, debuguj, debuguj 549

D

Dziesięć najważniejszych nieomówionych projektów

Jeszcze więcej narzędzi, bibliotek i modułów

Wiemy, co sobie myślisz, czytając tytuł tego rozdziału.

Dlaczego, u licha, nie zatytułowali poprzedniego dodatku: *Dwadzieścia najważniejszych tematów, których nie poruszyliśmy*? Dlaczego *kolejne* 10? W poprzednim dodatku ograniczyliśmy się do opisu rzeczy, które są wbudowane w samego Pythona (jako część „baterii w zestawie" tego języka). W tym dodatku zarzucamy naszą sieć znacznie dalej, omawiając szereg technologii, które są dostępne, *ponieważ* Python istnieje. Mamy tu do dyspozycji mnóstwo interesujących rzeczy, a ich krótki przegląd — podobnie jak miało to miejsce w poprzednim dodatku — z pewnością nie zrobi Ci *najmniejszej* krzywdy.

1. Alternatywy dla wiersza poleceń powłoki Pythona 552
2. Alternatywy dla środowiska IDLE 553
3. Jupyter Notebook: IDE dostępne z poziomu przeglądarki WWW 554
4. Przeprowadzanie analizy danych 555
5. Rozwiązania do tworzenia aplikacji WWW 556
6. Korzystanie z danych sieciowych 557
7. Więcej źródeł danych 558
8. Narzędzia programistyczne 559
9. Kivy: nasz wybór w kategorii „Najfajniejszy projekt wszech czasów" 560
10. Alternatywne implementacje 561

E

Zaangażowanie się

Społeczność Pythona

Python to coś znacznie więcej niż tylko doskonały język programowania.

To również wspaniała społeczność. Społeczność Pythona jest serdeczna, różnorodna, otwarta, przyjazna, hojna i zawsze chętnie dzieli się swoją wiedzą. Jesteśmy w szoku, że nikt nie pomyślał do tej pory, aby umieścić to wszystko na jakiejś laurce! A tak poważnie, programowanie w Pythonie nie sprowadza się wyłącznie do samego języka. Wokół niego wyrósł cały ekosystem, na który składają się doskonałe książki, blogi, serwisy WWW, konferencje, spotkania, grupy użytkowników i prawdziwe osobowości. W tym dodatku dokonamy przeglądu społeczności Pythona i wskażemy, co może Ci ona zaoferować. Nie siedź samotnie i nie programuj tylko dla siebie: **zaangażuj się**!

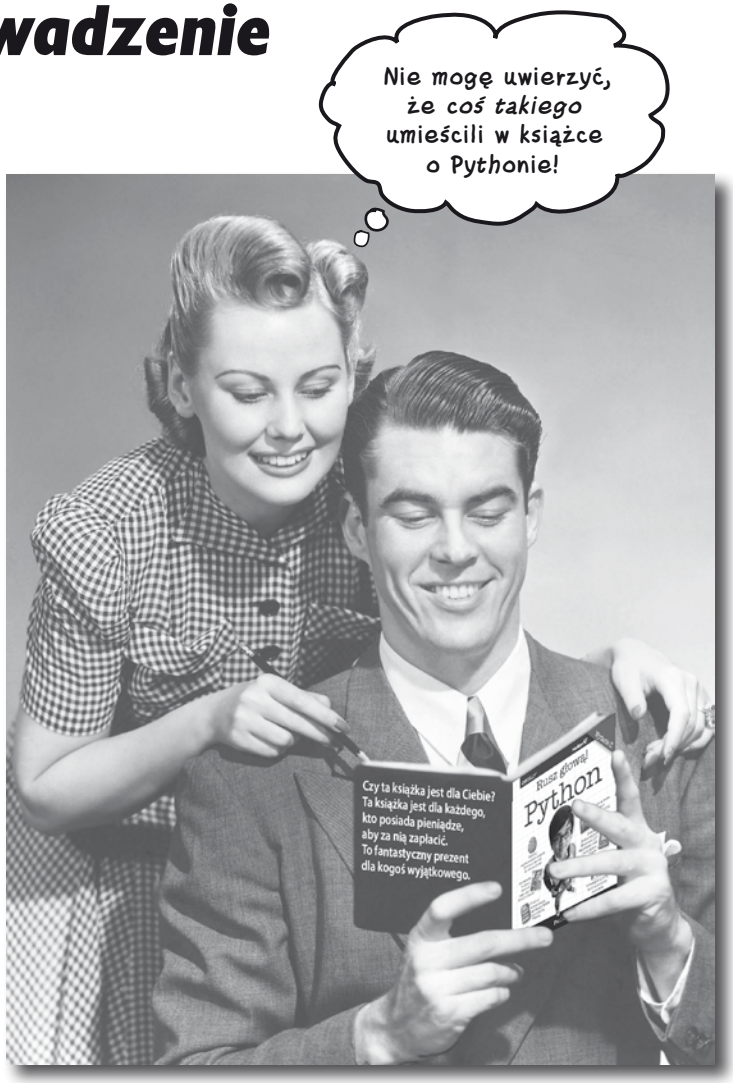
BDFL: Benevolent Dictator for Life	564
Tolerancyjna społeczność: szacunek dla różnorodności	565
Podkasty dotyczące Pythona	566
Zen Pythona	567
Którą książkę powinienem przeczytać jako następną?	568
Nasze ulubione książki na temat Pythona	569

S

Skorowidz 570

Jak korzystać z tej książki?

Wprowadzenie



W tym rozdziale odpowiadamy na palące pytanie: „Dlaczego autorzy UMIEŚCILI to wszystko w książce o języku Python?”.

Dla kogo jest ta książka?

Jeżeli możesz odpowiedzieć twierdząco na wszystkie poniższe pytania, książka ta jest dla Ciebie.

1. Czy umiesz już posługiwać się jakimś innym językiem programowania?
2. Czy chcesz dodać umiejętność programowania w Pythonie do listy opanowanych przez siebie narzędzi i wykorzystywać tę wiedzę do tworzenia nowych rzeczy?
3. Czy wolisz robić coś konkretnego i na bieżąco wykorzystywać przyswojone wiadomości w praktyce niż słuchać nudnych wykładów, z niecierpliwością oczekując końca zajęć?

Ta książka NIE jest materiałem źródłowym i zakładamy, że już wcześniej zetknąłeś się z programowaniem.

Kto raczej nie powinien sięgać po tę książkę?

Jeżeli możesz odpowiedzieć twierdząco na którekolwiek z poniższych pytań, ta książka ***nie*** jest dla Ciebie.

1. Czy masz już większość wiedzy niezbędnej do programowania w języku Python?
2. Czy szukasz dotyczącego Pythona materiału źródłowego, który opisuje wszystkie kwestie związane z tym językiem w najdrobniejszych szczegółach?
3. Czy wolałbyś raczej, aby 15 wrzeszczących małp wyrywało Ci paznokcie u nóg, niż nauczyć się czegoś nowego? Czy wierzysz, że książka poświęcona Pythonowi musi omawiać absolutnie wszystko, a jeśli przy okazji zanudza czytelnika do łez, to tylko lepiej dla niego?

[Uwaga z działu marketingu: ta książka jest dla każdego, kto ma kartę kredytową... Ale akceptujemy też przelewy].

Wiemy, co myślisz

Jakim cudem może *to* być poważna książka?

Po co te wszystkie obrazki?

Czy w ten sposób naprawdę mogę się czegokolwiek *nauczyć*?

Wiemy, co myśli Twój *mózg*

Twój mózg pragnie nowości. Zawsze szuka, przegląda i *wyczekuje* czegoś niezwykłego. Tak jest skonstruowany i dzięki temu masz szansę przeżyć.

Zatem, co Twój mózg robi z tymi wszystkimi rutynowymi, zwyczajnymi, normalnymi informacjami, jakie do niego docierają? Robi wszystko, co tylko *może*, aby nie przeszkadzały mu w jego *najważniejszym zadaniu* — zapamiętywaniu rzeczy, które mają *prawdziwe znaczenie*. Twój mózg nie traci czasu i energii na zapamiętywanie nudnych informacji; one nigdy nie przechodzą przez filtr odrzucający od razu wszystko, co jest „całkowicie nieistotne".

Skąd w takim razie Twój mózg *wie*, co jest istotne? Wyobraź sobie, że wędrujesz po lesie i nagle wyskakuje na Ciebie tygrys. Co się dzieje w Twojej głowie i Twoim ciele?

Neurony płoną. Emocje szaleją. *Adrenalina napływa falami.*

I właśnie dlatego Twój mózg wie, że…

To musi być ważne! Nie zapominaj o tym!

Wyobraź sobie teraz, że siedzisz w domu albo w bibliotece. Jest bezpiecznie i miło, a w okolicy nie grasują żadne tygrysy. Uczysz się. Przygotowujesz się do egzaminu. A może rozgryzasz jakiś trudny problem techniczny, którego rozwiązanie, według szefa, powinno zająć Ci tydzień, co najwyżej dziesięć dni.

Jest tylko jeden, drobny problem. Twój mózg stara się pomóc. Podejmuje próby, aby te w oczywisty sposób *nieistotne* informacje nie zajmowały cennych zasobów w Twojej głowie. Zasobów, które powinny zostać wykorzystane na zapamiętanie naprawdę *ważnych* rzeczy. Takich jak tygrysy. Takich jak zagrożenie, jakie niesie ze sobą pożar. Takich jak to, że nigdy nie powinieneś publikować na Facebooku zdjęć z imprezy. Co gorsza, nie ma żadnego sposobu, aby powiedzieć mózgowi: „Hej, mózgu mój, dziękuję ci bardzo, ale niezależnie od tego, jak nudna jest ta książka i jak nieznaczne są emocje, których aktualnie doznaję, to jednak *naprawdę* chciałbym zapamiętać wszystkie te informacje".

Wyobrażamy sobie, że Czytelnik tej książki jest uczniem

Zatem, czego trzeba, żeby się czegoś *nauczyć*? Najpierw powinieneś to coś *poznać*, a następnie postarać się, by tego czegoś nie *zapomnieć*. I nie chodzi tu jedynie o wtłoczenie do głowy suchych faktów. Najnowsze badania prowadzone w dziedzinie przyswajania informacji, neurobiologii i psychologii nauczania pokazują, że *uczenie się* wymaga czegoś więcej niż tylko czytania tekstu. My wiemy, co potrafi pobudzić nasze mózgi do działania.

Oto wybrane zasady obowiązujące autorów w książkach z serii Rusz głową!.

Wizualizuj. Obrazy znacznie łatwiej zapamiętać niż sam tekst i dzięki nim nauka staje się o wiele bardziej efektywna (można w ten sposób osiągnąć nawet 89-procentowy wzrost efektywności w przypominaniu i przekazywaniu informacji). Co więcej, rysunki sprawiają, że przekazywane informacje stają się bardziej zrozumiałe. Wystarczy **umieścić słowa bezpośrednio na rysunku (lub w jego okolicy)** zamiast na następnej stronie, a prawdopodobieństwo, że osoby uczące się będą w stanie rozwiązać problem, którego te słowa dotyczą, wzrośnie nawet *dwukrotnie*.

Stosuj gawędziarski i spersonifikowany styl. Według najnowszych badań w testach końcowych studenci uzyskiwali wyniki o 40% lepsze, jeśli treść była przekazywana w sposób bezpośredni, w pierwszej osobie i w konwencji rozmowy, a nie w sposób formalny. Zamiast wykładać, opowiadaj historyjki. Używaj języka potocznego. Nie traktuj siebie zbyt poważnie. Kiedy byłbyś bardziej uważny: podczas stymulującej rozmowy przy obiedzie czy podczas wykładu?

Zachęć czytelników do głębszych przemyśleń. Innymi słowy, jeśli nie zmusisz neuronów do aktywnego wysiłku, w głowie nie zdarzy się nic wielkiego. Czytelnik musi być zmotywowany, zaangażowany, zaciekawiony i podekscytowany rozwiązywaniem problemów, wyciąganiem wniosków i zdobywaniem nowej wiedzy. Osiągnięcie tego wszystkiego jest możliwe poprzez stawianie wyzwań, zapraszanie do wykonywania ćwiczeń i zadawanie pytań zmuszających do zastanowienia oraz poprzez nakłanianie do działań, które wymagają zaangażowania obu półkul mózgowych i wielu zmysłów.

Zdobądź oraz przyciągnij na dłużej uwagę i zainteresowanie czytelnika. Każdy znalazł się kiedyś w sytuacji, gdy bardzo chciał się czegoś nauczyć, lecz zasypiał po przeczytaniu pierwszej strony. Mózg zwraca uwagę na rzeczy niezwykłe, interesujące, dziwne, przykuwające wzrok, nieoczekiwane. Jednak poznawanie nowego zagadnienia technicznego wcale nie musi być nudne. Jeśli będzie to zagadnienie interesujące, Twój mózg przyswoi je znacznie szybciej.

Wyzwól emocje. Teraz już wiemy, że zdolności do zapamiętywania informacji są w znacznej mierze zależne od ich zawartości emocjonalnej. Zapamiętujemy to, na czym nam zależy. Zapamiętujemy w sytuacjach, w których coś *odczuwamy*. Oczywiście nie mamy tu na myśli wzruszających historii o chłopcu i jego psie. Chodzi o takie emocje jak zaskoczenie, ciekawość, radosne podekscytowanie, „a niech to..." i uczucie satysfakcji — „Jestem wielki!" — jakie odczuwamy po rozwiązaniu zagadki, *nauczeniu się* czegoś, co powszechnie uchodzi za trudne, lub zdaniu sobie sprawy, że *znamy* więcej szczegółów technicznych niż Robert z działu inżynierii.

Metapoznanie — myślenie o myśleniu

Jeśli naprawdę chcesz się czegoś nauczyć, a dodatkowo chcesz się tego nauczyć szybciej i bardziej dogłębnie, zwracaj uwagę na to, jak Ci na tym zależy. Pomyśl o tym, jak myślisz. Poznaj sposób, w jaki się uczysz.

Większość z nas w okresie dorastania nie uczestniczyła w zajęciach z metapoznania albo teorii nauczania. *Oczekiwano* od nas, że będziemy się uczyć, jednak *nie uczono* nas, jak mamy to robić.

Zakładamy jednak, że jeśli trzymasz w ręku tę książkę, chcesz nauczyć się rozwiązywać problemy za pomocą Pythona. I prawdopodobnie nie zamierzasz na to tracić zbyt wiele czasu. Jeśli chcesz wykorzystać to, co przeczytałeś w tej książce, musisz to *zapamiętać*. A oprócz tego musisz to *zrozumieć*. Aby w możliwie jak największym stopniu wykorzystać zarówno tę, jak i *dowolną* inną książkę lub jakikolwiek inny sposób uczenia się, musisz wziąć odpowiedzialność za swój mózg. Myśl o *tym*, czego się uczysz.

Sztuczka polega na tym, aby przekonać mózg, że poznawany materiał jest *naprawdę ważny*. Kluczowy dla Twojego dobrego samopoczucia. Tak ważny jak tygrys stojący naprzeciw Ciebie. W przeciwnym razie będziesz prowadzić nieustającą wojnę z własnym mózgiem, który ze wszystkich sił będzie się starać, aby nowe informacje nie zostały utrwalone.

W jaki sposób ZMUSIĆ mózg, aby potraktował programowanie jak głodnego tygrysa?

Można to zrobić w sposób wolny i męczący lub szybki i bardziej efektywny. Wolny sposób polega na wielokrotnym powtarzaniu. Oczywiście wiesz, że *jesteś* w stanie nauczyć się i zapamiętać nawet najnudniejsze zagadnienie, mozolnie je „wkuwając". Po odpowiedniej liczbie powtórzeń Twój mózg stwierdzi: „*Wydaje* się, że to nie jest dla niego szczególnie ważne, lecz w *kółko* to czyta i powtarza, więc przypuszczam, że jakąś wartość to jednak musi mieć".

Szybszy sposób polega na zrobieniu ***czegokolwiek, co zwiększy aktywność mózgu***, zwłaszcza wtedy, kiedy czynność ta wyzwoli kilka różnych *typów* jego aktywności. Wszystkie zagadnienia, o których pisaliśmy na poprzedniej stronie, są kluczowymi elementami rozwiązania i udowodniono, że wszystkie one mogą pomóc w zmuszeniu mózgu do tego, aby pracował na Twoją korzyść. Przykładowo, badania wykazują, że umieszczenie słów *w obrębie* opisywanych rysunków (a nie w innych miejscach tekstu na stronie, na przykład w nagłówku lub wewnątrz akapitu) sprawia, że mózg stara się zrozumieć relację pomiędzy słowami a rysunkiem, a to z kolei zwiększa aktywność neuronów. Większa aktywność neuronów to większa szansa, że mózg *uzna* informacje za warte zainteresowania i — ewentualnie — zapamiętania.

Prezentowanie informacji w formie konwersacji pomaga, gdyż ludzie zdają się wykazywać większe zainteresowanie, kiedy uważają, że biorą udział w rozmowie i oczekuje się od nich, iż będą śledzić jej przebieg i brać w niej czynny udział. Zadziwiające jest to, iż mózg zdaje się nie *zważać*, że rozmowa jest prowadzona z książką! Z drugiej strony, jeśli sposób przedstawiania informacji jest formalny i suchy, mózg postrzega to tak samo jak w sytuacji, gdy uczestniczysz w wykładzie na sali pełnej sennych studentów: nie trzeba wykazywać żadnej aktywności.

Jednak rysunki i przedstawianie informacji w formie rozmowy to tylko początek…

To, co MY zrobiliśmy

Używaliśmy ***rysunków***, ponieważ Twój mózg zwraca większą uwagę na obrazy niż na tekst. Dla mózgu faktycznie jeden obraz *jest* wart tysiąca słów. W sytuacjach, gdy pojawiał się zarówno tekst, jak i rysunek, umieszczaliśmy tekst *na* rysunku, gdyż mózg działa bardziej efektywnie, gdy tekst jest *wewnątrz* czegoś, co opisuje, niż kiedy jest umieszczony w innym miejscu i stanowi część większego fragmentu.

Stosowaliśmy ***powtórzenia***, wielokrotnie podając tę samą informację na *różne* sposoby i przy wykorzystaniu różnych środków przekazu oraz odwołując się do *różnych zmysłów*. Wszystko po to, aby zwiększyć szansę, że informacja zostanie zakodowana w większej liczbie obszarów Twojego mózgu.

Używaliśmy pomysłów i rysunków w ***nieoczekiwany*** sposób, ponieważ Twój mózg oczekuje i pragnie nowości, poza tym staraliśmy się zawrzeć w nich *chociaż trochę* ***emocji***, gdyż mózg jest skonstruowany w taki sposób, iż zwraca uwagę na biochemię związaną z emocjami. Prawdopodobieństwo zapamiętania czegoś jest większe, jeśli „to coś" sprawia, że coś *poczujemy*, nawet jeśli to uczucie nie jest niczym więcej niż lekkim ***rozbawieniem***, ***zaskoczeniem*** lub ***zainteresowaniem***.

Używaliśmy bezpośrednich zwrotów i przekazywaliśmy treści w ***formie konwersacji***, gdyż mózg zwraca większą uwagę, jeśli sądzi, że prowadzisz rozmowę, niż gdy jesteś jedynie biernym słuchaczem prezentacji. Mózg działa w ten sposób nawet wtedy, gdy *czytasz* zapis rozmowy.

Zamieściliśmy w książce ponad 80 ***ćwiczeń***, ponieważ mózg uczy się i pamięta więcej, gdy coś ***robi***, niż gdy o czymś *czyta*. Ćwiczenia te stanowią wyzwania, choć nie są przesadnie trudne, gdyż właśnie takie preferuje większość osób.

Stosowaliśmy ***wiele stylów nauczania***, gdyż *Ty* możesz preferować instrukcje opisujące krok po kroku sposób postępowania, ktoś inny — analizowanie zagadnienia opisanego w ogólny sposób, a jeszcze inne osoby — przejrzenie przykładowego fragmentu kodu. Jednak niezależnie od ulubionego sposobu nauki *każdy* skorzysta na tym, że te same informacje będą przedstawiane kilkakrotnie na różne sposoby.

Naprzemiennie podawaliśmy informacje przeznaczone dla ***obu półkul Twojego mózgu***, gdyż im bardziej mózg jest zaangażowany, tym większe prawdopodobieństwo nauczenia się i zapamiętania podawanych informacji i tym dłużej możesz koncentrować się na nauce. Ponieważ angażowanie tylko jednej półkuli mózgu w danej chwili często oznacza, że druga może odpocząć, zatem możesz uczyć się bardziej produktywnie przez dłuższy czas.

Dodatkowo zamieszczaliśmy ***historyjki*** i ćwiczenia prezentujące ***więcej niż jeden punkt widzenia***, ponieważ mózg uczy się łatwiej, gdy jest zmuszony do przetwarzania i podawania własnej opinii.

Stawialiśmy przed Tobą ***wyzwania***, zarówno proponując ćwiczenia, jak i formułując ***pytania***, na które nie zawsze można odpowiedzieć w prosty sposób, a to dlatego, że mózg uczy się i pamięta, gdy musi nad czymś *popracować*. Pomyśl sam — nie można zyskać dobrej *kondycji*, jedynie *obserwując* ludzi ćwiczących na sali gimnastycznej. Jednak dołożyliśmy wszelkich starań, aby zapewnić, że gdy pracujesz, robisz *dokładnie* to, co trzeba. Postaraliśmy się, aby ***ani jeden dendryt nie musiał*** przetwarzać przykładu zbyt trudnego ani analizować tekstu zbyt lapidarnego lub napisanego niezrozumiałym żargonem.

Przedstawialiśmy ***ludzi***. Robiliśmy to w historyjkach, przykładach, na rysunkach itd., gdyż *Ty też jesteś* człowiekiem. Dlatego Twój mózg zwraca więcej uwagi na *ludzi* niż na *rzeczy*.

Wytnij te porady i przyklej je na lodówce.

To, co TY możesz zrobić, aby zmusić swój mózg do posłuszeństwa

A zatem zrobiliśmy, co było w naszej mocy. Reszta zależy od Ciebie. Możesz zacząć od poniższych porad. Posłuchaj swojego mózgu i określ, które sposoby sprawdzają się w Twoim przypadku, a które nie dają pozytywnych rezultatów. Wypróbuj kilka z nich.

1. **Zwolnij. Im więcej rozumiesz, tym mniej musisz zapamiętać.**
   Nie ograniczaj się jedynie do *czytania*. Przerwij na chwilę lekturę i pomyśl. Kiedy znajdziesz w tekście pytanie, nie zaglądaj od razu na stronę z odpowiedzią. Wyobraź sobie, że ktoś faktycznie *zadaje* Ci pytanie. Im bardziej zmusisz swój mózg do myślenia, tym większa będzie szansa, że się czegoś nauczysz i coś zapamiętasz.

2. **Wykonuj ćwiczenia. Rób notatki.**
   Umieszczaliśmy je w tekście, jeśli jednak zrobilibyśmy je za Ciebie, niczym nie różniłoby się to od sytuacji, w której ktoś za Ciebie wykonywałby ćwiczenia fizyczne. I nie ograniczaj się jedynie do *czytania* ćwiczeń. **Używaj ołówka**. Można znaleźć wiele dowodów na to, że fizyczna aktywność *podczas* nauki może poprawić jej wyniki.

3. **Czytaj fragmenty oznaczone jako „Nie istnieją głupie pytania".**
   Chodzi tu o wszystkie fragmenty tego rodzaju. Nie są to fragmenty opcjonalne — stanowią ***część podstawowej treści książki***! Nie pomijaj ich.

4. **Niech lektura tej książki będzie ostatnią rzeczą, jaką robisz przed pójściem spać. A przynajmniej ostatnią rzeczą stanowiącą wyzwanie intelektualne.**
   Pewne elementy procesu uczenia się (a szczególnie przenoszenie informacji do pamięci długoterminowej) następują *po* odłożeniu książki. Twój mózg potrzebuje trochę czasu dla siebie i musi dodatkowo przetworzyć dostarczone informacje. Jeśli w tym czasie, koniecznym na wykonanie dodatkowego „przetwarzania", zmusisz go do innej działalności, część z przyswojonych informacji może zostać utracona.

5. **Rozmawiaj o zdobywanych informacjach. Głośno.**
   Mówienie aktywuje odmienne fragmenty mózgu. Jeśli próbujesz coś zrozumieć lub zwiększyć szansę na zapamiętanie informacji na dłużej, powtarzaj je na głos. Jeszcze lepiej — staraj się je głośno komuś wytłumaczyć. W ten sposób nauczysz się szybciej, a oprócz tego będziesz mógł odkryć kwestie, o których nie wiedziałeś podczas czytania tekstu książki.

6. **Pij wodę. Dużo wody.**
   Twój mózg pracuje najlepiej, gdy dostarcza mu się dużo płynów. Odwodnienie (które może następować nawet wtedy, zanim poczujesz pragnienie) obniża zdolność percepcji.

7. **Posłuchaj swojego mózgu.**
   Uważaj, kiedy Twój mózg staje się przeciążony. Jeśli spostrzeżesz, że zaczynasz czytać pobieżnie i zapominać to, o czym przeczytałeś przed chwilą, to najwyższy czas zrobić przerwę. Po przekroczeniu pewnego punktu nie będziesz się uczył szybciej, „wciskając" do głowy więcej informacji; co gorsza, może to zaszkodzić całemu procesowi nauki.

8. **Poczuj coś.**
   Twój mózg musi wiedzieć, że to, czego się uczysz, ma *znaczenie*. Z zaangażowaniem śledź zamieszczane w tekście historyjki. Nadawaj zdjęciom własne tytuły. Załamywanie rąk po przeczytaniu głupiego dowcipu *i tak jest lepsze* od braku jakiejkolwiek reakcji.

9. **Pisz jak najwięcej kodu!**
   Jest tylko jeden sposób umożliwiający nauczenie się programowania w Pythonie i polega on na **pisaniu mnóstwa kodu**. I właśnie to będziesz robił, czytając tę książkę. Programowanie to umiejętność, a jedyną metodą, aby ją posiąść na odpowiednim poziomie, jest praktyka. Zamierzamy dać Ci jej naprawdę dużo; w każdym rozdziale znajdują się ćwiczenia, które stawiają przed Tobą problem do rozwiązania. Nie pomijaj ich ot tak, po prostu, ponieważ znaczna część nauki odbywa się właśnie podczas wykonywania ćwiczeń. Do każdego z nich zapewniliśmy rozwiązanie. Nie wahaj się **do nich zaglądać**, gdy z czymś utkniesz! (Łatwo jest dać się złapać jakiejś drobnostce). Zanim jednak spojrzysz na rozwiązanie problemu, spróbuj go rozwiązać. I zdecydowanie, spraw, żeby Twoje rozwiązanie działało, zanim przejdziesz do następnego fragmentu książki.

Przeczytaj to (1. z 2)

Książka ta jest doznaniem poznawczym, a nie podręcznikiem. Celowo usunęliśmy z niej wszystko, co mogłoby Ci przeszkadzać w uczeniu się i poznawaniu materiału zamieszczonego w jej konkretnym miejscu. Podczas pierwszej lektury tej książki powinieneś zaczynać od samego początku, gdyż jej dalsze fragmenty bazują na wiedzy, którą powinieneś zdobyć wcześniej.

Celem tej książki jest jak najszybsze wdrożenie Cię do pracy.

Uczymy Cię określonych rzeczy wtedy, gdy są Ci potrzebne. Z tego powodu nie znajdziesz tu długich list znanych z materiałów technicznych, tabel zawierających operatory Pythona ani reguł pierwszeństwa operatorów. Nie opisujemy *wszystkiego*, lecz w jak największym stopniu skupiamy się na przedstawieniu podstawowego materiału, aby Twój mózg mógł *szybko* przyswoić Pythona i na długo go w sobie zatrzymać. Jedynym założeniem, które przyjęliśmy, jest to, że wiesz już, jak programować w jakimś innym języku.

Książka opisuje język Python 3.

W książce tej korzystamy z wersji 3 języka programowania Python, a w dodatku A opisujemy, jak ją pobrać i zainstalować. W książce **nie** jest używany Python 2.

Od razu zaganiamy Pythona do pracy.

Już w rozdziale 1. angażujemy Cię w tworzenie pożytecznych rozwiązań, na których podstawie budujemy kolejne aplikacje w dalszym ciągu książki. Nie kręcimy się tu bez celu, ponieważ chcemy, aby Twoja praca była od razu *produktywna*.

Aktywności NIE są opcjonalne — musisz wykonać odpowiednią pracę.

Ćwiczenia i aktywności *nie są* jedynie dodatkami, są elementami treści tej książki. Niektóre z nich mają wspomóc Twoją pamięć, inne — ułatwić zrozumienie, a jeszcze inne — pomóc w wykorzystaniu nabytej wiedzy i umiejętności. ***Nie pomijaj tych ćwiczeń.***

Powtórzenia są celowe i ważne.

Jedną z cech wyróżniających książki z serii *Rusz głową!* jest to, że nam *naprawdę* zależy, żebyś wszystko zrozumiał. Chcemy także, byś kończąc lekturę tej książki, pamiętał wszystko, co w niej przeczytałeś. Większość podręczników i wydawnictw encyklopedycznych nie zwraca uwagi na przyswojenie i zapamiętanie informacji, jednak w tej chodzi o *naukę*, dlatego znajdziesz w niej wiele pojęć, które pojawiają się kilka razy.

Przykładowe kody są jak najbardziej zwięzłe.

Czytelnicy niejednokrotnie zwracali nam uwagę, że przeglądanie 200 wierszy kodu w poszukiwaniu dwóch linijek, które należy zrozumieć, może być frustrujące. Kod większości przykładów zamieszczonych w tej książce został w jak największym stopniu skrócony; dzięki temu fragmenty, których musisz się nauczyć, są przejrzyste i proste. Nie należy zatem oczekiwać, że podawane przykłady będą solidne ani nawet kompletne — zostały one opracowane wyłącznie pod kątem *nauki*, a nie zapewnienia pełnej funkcjonalności (choć staraliśmy się ją w jak największym stopniu zapewnić).

Przeczytaj to (2. z 2)

Tak, jest tego więcej...

Drugie wydanie książki NIE przypomina pierwszego.

Ta książka to aktualizacja pierwszego wydania *Python. Rusz głową!*, które zostało opublikowane pod koniec 2010 roku. Choć tamta książka i ta, którą trzymasz w rękach, mają tego samego autora, jest on już starszy i (miejmy nadzieję) mądrzejszy niż wtedy. W związku z tym postanowił napisać na nowo treść pierwszego wydania. Dlatego *wszystko* jest tu nowe: kolejność rozdziałów jest inna, treść została zaktualizowana, przykłady są lepsze, a historyjki zostały usunięte lub zastąpione innymi. Pozostaliśmy przy tej samej okładce, choć z pewnymi, niewielkimi zmianami, ponieważ stwierdziliśmy, że nie chcemy aż tak bardzo kołysać tą łodzią. Od wydania pierwszej edycji książki minęło sześć długich lat. Mamy nadzieję, że spodoba Ci się to, co udało się nam osiągnąć w tym czasie.

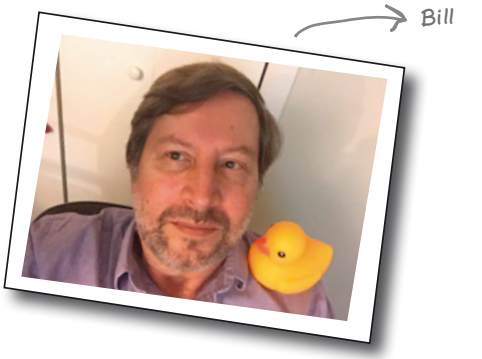
Gdzie jest kod?

Wszystkie przykłady prezentowane w tej książce umieściliśmy w internecie, dzięki czemu będziesz mógł skopiować je i wkleić, gdy tylko okaże się to potrzebne (choć zalecamy Ci wprowadzanie ich ręcznie, *gdy będziesz podążał za prezentowaną przez nas treścią*). Kody możesz pobrać z serwera FTP wydawnictwa Helion:

ftp://ftp.helion.pl/przyklady/pytgr2.zip

Recenzenci techniczni

Bill Lubanovic jest programistą i administratorem od 40 lat. Pisał również dla wydawnictwa O'Reilly: jest autorem rozdziałów dotyczących bezpieczeństwa, które wchodzą w skład książek na temat systemu Linux, współautorem książki poświęconej administrowaniu systemem Linux, a także autorem książki *Introducing Python*. Bill mieszka nad lodowatym jeziorem w górach Sangre de Sasquatch w Minnesocie z jedną cudowną żoną, dwojgiem cudownych dzieci i trzema futrzastymi kotami.



Edward Yue Shung Wong jest pochłonięty programowaniem od czasu, gdy w 2006 roku napisał swój pierwszy wiersz kodu w języku Haskell. Obecnie pracuje nad sterowanym zdarzeniami tradeprocessingiem w samym centrum londyńskiego City. Edward lubi dzielić się swoją pasją do programowania z członkami organizacji London Java Community oraz Software Craftsmanship Community. Gdy jest z dala od swojej klawiatury, można go znaleźć w jego żywiole, którym jest boisko do piłki nożnej oraz poświęcony grom komputerowym kanał w serwisie YouTube (*@arkangelofkaos*).

Adrienne Lowe to była osobista kucharka z Atlanty przemieniona w programistkę Pythona, która dzieli się ciekawymi historiami, podsumowaniami konferencji oraz przepisami na dania na swoim blogu o nazwie *Coding with Knives* (*http://codingwithknives.com*) poświęconym gotowaniu i programowaniu. Organizuje spotkania PyLadiesATL oraz Django Girls Atlanta, a także odpowiada za serię cotygodniowych wywiadów pod hasłem *Your Django Story* przeprowadzanych w ramach Django Girls dla kobiet używających języka Python. Adrienne pracuje na stanowisku inżyniera wsparcia technicznego w firmie Emma Inc. oraz jako dyrektor ds. rozwoju w Django Software Foundation, należy też do podstawowego zespołu Write the Docs. Woli tradycyjne, ręcznie pisane listy od e-maili i od dzieciństwa tworzy swoją kolekcję znaczków pocztowych.

Monte Milanuk dostarczył wartościowych informacji zwrotnych.

Podziękowania

Dawn

Moja redaktor: Redaktorem tego wydania książki jest **Dawn Schanafelt**; dzięki jej zaangażowaniu książka ta jest o wiele, wiele lepsza, niż byłaby bez niego. Dawn jest nie tylko wspaniałym redaktorem, lecz również ma świetne oko do szczegółów oraz doskonale potrafi wyrażać swoje myśli, co w dużej mierze poprawiło przedstawioną tu treść. Firma *O'Reilly Media* ma wspaniały zwyczaj zatrudniania bystrych, przyjaznych oraz kompetentnych osób, a Dawn jest uosobieniem wszystkich tych cech.

Zespół O'Reilly Media: Napisanie tego wydania książki *Python. Rusz głową!* zajęło cztery lata (to długa historia). Nic dziwnego zatem, że w jej powstanie zaangażowanych było wiele osób należących do zespołu *O'Reilly Media*. **Courtney Nash** przekonała mnie do „szybkiego przepisania" tej książki w 2012 roku, a następnie była pod ręką, gdy zakres projektu gwałtownie się rozrósł. Courtney była pierwszą redaktor tego wydania książki; wspierała mnie, gdy zdarzyła się katastrofa i wydawało się, że cały ten projekt został skazany na niepowodzenie. Gdy sprawy zaczęły powoli wracać do normy, Courtney zajęła się większymi i lepszymi rzeczami w *O'Reilly Media*, przekazując obowiązki redakcyjne w 2014 roku bardzo zajętej **Meghan Blanchette**, która obserwowała (jak sądzę, z rosnącym przerażeniem), jak opóźnienie narasta, a książka ta wraca na właściwe tory, po czym zjeżdża z nich w regularnych odstępach czasu. Sprawy dopiero co zaczęły znów wracać do normy, gdy Meghan wyruszyła na nowe pastwiska, a obowiązki redaktora tej książki przejęła Dawn. Było to rok temu, a większość z należących do tej publikacji 12 3/4 rozdziałów została napisana pod zawsze czujnym okiem Dawn. Jak już wcześniej wspomniałem, *O'Reilly Media* zatrudnia bardzo dobrych ludzi, a wkład redakcyjny wniesiony przez Courtney i Meghan oraz udzielane przez nie wsparcie zasługują na najwyższe uznanie. Poza nimi podziękowania należą się **Maureen Spencer**, **Heather Scherer**, **Karen Shaner** oraz **Chrisowi Pappasowi** za pracę „za kulisami". Dziękuję również wszystkim niewidzialnym, niedocenianym i niewymienionym z nazwiska bohaterom, znanym jako **Produkcja**, którzy wzięli moje opracowane w *InDesignie* rodziały i przekształcili je w gotowy produkt, który trzymasz teraz w rękach. Wykonali naprawdę świetną robotę.

Wyrazy wdzięczności należą się też **Bertowi Batesowi**, który wraz z **Kathy Sierrą** opracował tę serię książek, w tym doskonałą publikację *Java. Rusz głową!*. Bert poświęcił mnóstwo czasu, pracując ze mną w celu zapewnienia, żeby to wydanie zostało skierowane we właściwą stronę.

Przyjaciele i współpracownicy: Raz jeszcze bardzo dziękuję **Nigelowi Whyte'owi** (kierownikowi *Katedry Informatyki Instytutu Technologicznego* w *Carlow*) za wsparcie mojego zaangażowania w napisanie tej książki na nowo. Wielu moich studentów zostało zmuszonych do przerobienia sporej części tego materiału w ramach swoich studiów i mam nadzieję, że chichoczą na widok wydrukowanego w książce przykładu, z którym mieli do czynienia na zajęciach (lub nawet wielu takich przykładów).

Raz jeszcze dziękuję **Davidowi Griffithsowi** (mojemu wspólnikowi zbrodni na publikacji *Head First Programming*) za powiedzenie mi w jednym szczególnie złym punkcie, abym przestał rozdzierać szaty nad każdym szczegółem i po prostu wreszcie *napisał tę cholerną książkę*! Okazało się to doskonałą radą i dobrze jest wiedzieć, że David wraz z Down (będącą jego żoną i współautorką książek z serii *Rusz głową!*) jest zawsze tylko o e-mail stąd. Koniecznie zapoznaj się ze świetnymi książkami z serii *Rusz głową!* napisanymi przez Davida i Dawn.

Rodzina: Moja rodzina (żona **Deirdre** oraz dzieci: **Joseph**, **Aaron** i **Aideen**) musiała przez lata znosić wszelkie wzloty i upadki, zrywy i stagnacje oraz rozmaite zmieniające życie doświadczenia, z których wszyscy wyszliśmy obronną ręką i — na szczęście — o zdrowych zmysłach. Przetrwała ta książka, przetrwałem ja i przetrwała nasza rodzina. Jestem im wszystkim bardzo wdzięczny i bardzo ich kocham, a także wiem, że nie muszę tego pisać, ale napiszę: *robię to dla Was*.

Lista bez-kogo: Mój zespół recenzentów technicznych sprawił się doskonale: sprawdź ich miniprofile zamieszczone na poprzedniej stronie. Uwzględniłem wszystkie uwagi, które mi przekazali, poprawiłem wszystkie błędy, które znaleźli, i zawsze niezmiernie cieszyło mnie, gdy którekolwiek z nich poświęciło chwilę na to, aby powiedzieć mi, że podoba mu się to, co robię. Jestem im wszystkim bardzo wdzięczny.

1. Podstawy

Szybki start

Rozpocznij przygodę z Pythonem najszybciej, jak się da.

W tym rozdziale przedstawimy podstawy programowania w języku Python i zrobimy to w sposób typowy dla serii *Rusz głową!*: od razu wskakując w sam środek zagadnienia. Już po przeczytaniu kilku pierwszych stron uruchomisz swój pierwszy przykładowy program. Gdy dojdziesz do końca rozdziału, będziesz w stanie nie tylko uruchomić program, lecz również zrozumieć jego kod (i wiele więcej). W międzyczasie dowiesz się, co decyduje o charakterze **Pythona** jako języka programowania. Nie traćmy już zatem więcej czasu. Przewróć stronę i ruszaj w drogę!

Zrywanie z tradycją

Weź do ręki dowolną książkę na temat jakiegokolwiek języka programowania, a pierwszą rzeczą, którą ujrzysz, będzie przykład *Witaj, świecie*.

Nie, nie zaczniemy.

To książka z serii *Rusz głową!* i wszystko robimy tu trochę inaczej. W innych książkach tradycją jest na samym początku zaprezentowanie czytelnikowi sposobu tworzenia programu *Witaj, świecie* w omawianym języku. Jednak w przypadku Pythona sprowadza się on do pojedynczej instrukcji wywołującej wbudowaną funkcję `print`, która wyświetla na ekranie tradycyjny komunikat „Witaj, świecie!". To chyba aż nazbyt ekscytujące... i nie nauczy Cię prawie niczego.

Dlatego nie, nie mamy zamiaru pokazywać Ci programu *Witaj, świecie* w Pythonie, ponieważ nie ma w nim niczego, co poszerzyłoby Twoją wiedzę o języku. Zamierzamy skorzystać z innej ścieżki...

Bardziej treściwy przykład na początek

Planujemy zacząć ten rozdział od przykładu, który jest nieco bardziej rozbudowany i — co za tym idzie — bardziej użyteczny niż *Witaj, świecie*.

Nie zamierzamy ściemniać i od razu powiemy Ci, jak będzie. Nasz przykład jest dość wydumany: wykonuje pewne działanie, ale nie okaże się zbyt przydatny w dłuższej perspektywie. Mamy tu na myśli to, że wybraliśmy go jako środek umożliwiający przedstawienie dużej części możliwości języka Python w jak najkrótszym czasie. Obiecujemy przy tym, że gdy skończysz się już zajmować tym przykładem, będziesz wiedział wystarczająco wiele, aby poradzić sobie z napisaniem w Pythonie programu *Witaj, świecie* bez naszej pomocy.

Wskocz prosto do środka

Jeśli nie zainstalowałeś jeszcze na swoim komputerze Pythona 3, przerwij na chwilę lekturę i przejdź prosto do dodatku A, w którym znajdziesz prowadzącą Cię krok po kroku instrukcję instalacji (obiecujemy, że cały proces zajmie Ci dosłownie kilka minut).

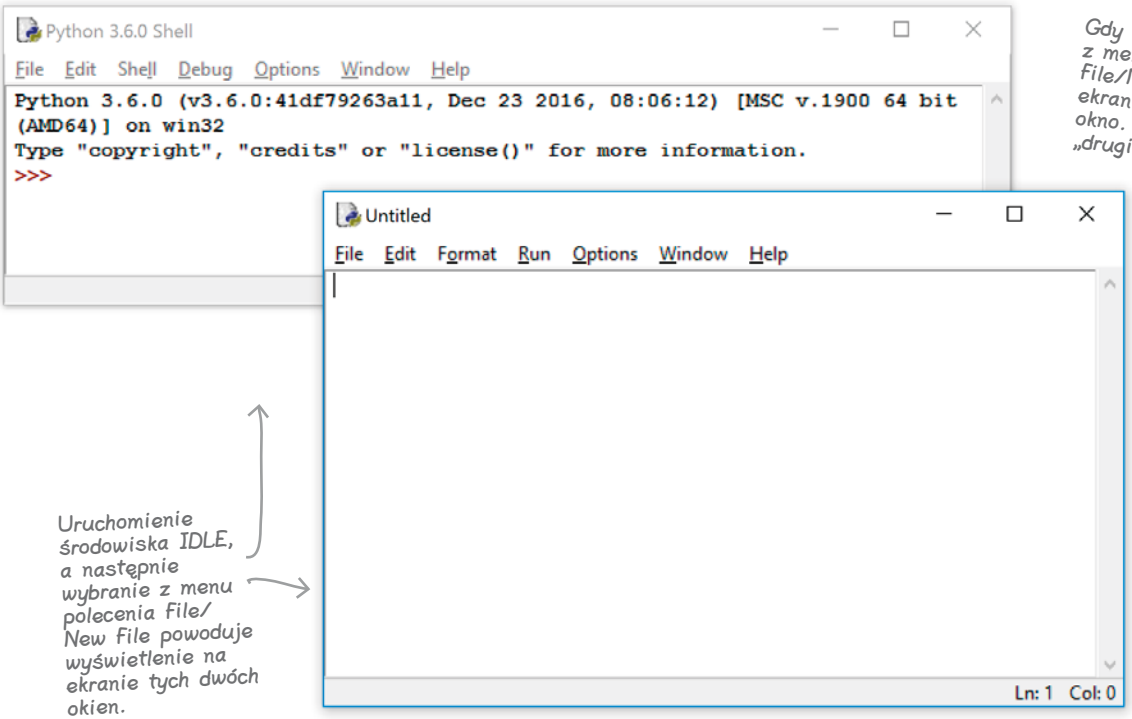
Dysponując najnowszą wersją Pythona 3, będziesz gotów do rozpoczęcia programowania w tym języku. Aby było Ci łatwiej, będziesz na razie korzystać z wbudowanego zintegrowanego środowiska programistycznego (ang. *integrated development environment*, *IDE*).

IDLE jest wszystkim, czego potrzebujesz, aby ruszyć

Gdy na swoim komputerze zainstalujesz Pythona 3, w pakiecie dostaniesz też bardzo proste, lecz użyteczne, zintegrowane środowisko programistyczne o nazwie IDLE. Choć istnieje wiele różnych sposobów umożliwiających uruchamianie kodu Pythona (i z wieloma z nich zapoznasz się w tej książce), IDLE stanowi wszystko, czego potrzebujesz na starcie.

Uruchom IDLE na swoim komputerze, a następnie skorzystaj z polecenia menu *File/New File* (plik/nowy plik), aby otworzyć nowe okno edycji. Gdy zrobiliśmy to na naszym komputerze, na ekranie mieliśmy już dwa okna: jedno o nazwie *Python Shell* (powłoka Pythona) i drugie o nazwie *Untitled* (bez nazwy).

Najpierw wyświetlone zostanie to okno. Traktuj je jako „pierwsze okno".



Gdy wybierzesz z menu polecenie File/New File, na ekranie pojawi się to okno. Traktuj je jako „drugie okno".

Uruchomienie środowiska IDLE, a następnie wybranie z menu polecenia File/New File powoduje wyświetlenie na ekranie tych dwóch okien.

Znaczenie okien środowiska IDLE

Obydwa te okna IDLE są ważne.

Pierwsze z nich, *Python Shell*, to środowisko REPL wykorzystywane do uruchamiania fragmentów kodu Pythona, przeważnie pojedynczych instrukcji. Im dłużej będziesz korzystał z tego języka, tym bardziej będziesz kochał powłokę Pythona, a w czasie lektury tej książki będziesz jej używał naprawdę intensywnie. Teraz jednak bardziej interesuje nas drugie z okien.

Drugie okno, *Untitled*, to okno edycji tekstu, którego można używać do pisania całych programów w Pythonie. Edytor środowiska IDLE nie jest najwspanialszym narzędziem tego typu na świecie (ponieważ ten tytuł należy się *<tu wpisz nazwę swojego ulubionego edytora>*), ale jest całkiem użyteczny i oferuje sporo nowoczesnych funkcji wbudowanych, w tym wyróżnianie składni za pomocą kolorów i temu podobne.

Ponieważ od razu wskakujemy w temat, zrób odważnie krok do przodu i napisz w tym oknie niewielki program w języku Python. Gdy skończysz wprowadzanie przedstawionego poniżej kodu, użyj polecenia menu *File/Save* (plik/zapisz), aby zapisać swój program pod nazwą *odd.py*.

Pamiętaj, aby wprowadzić kod *dokładnie* tak, jak pokazano poniżej.

Dla maniaków

Co oznacza REPL?

To skrót, którego używają maniacy, gdy mówią o czytaniu, ewaluacji, wyświetlaniu i powtarzaniu (ang. *read-eval-print-loop*). Opisuje on interaktywne narzędzie programistyczne, które pozwala dowolnie eksperymentować z fragmentami kodu. Więcej informacji na ten temat znajdziesz pod adresem *https://pl.wikipedia.org/wiki/REPL* (a jeśli tego Ci mało i nie boisz się angielskiego, zerknij tutaj: *http://en.wikipedia.org/wiki/Read-eval-print_loop*).

odd.py - C:/Users/lsuma/Desktop/Python/R01/odd.py (3.6.0)

File Edit Format Run Options Window Help

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

Ln: 14 Col: 0

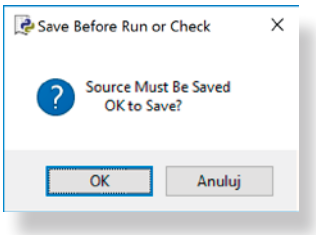
Nie przejmuj się teraz tym, co robi ten kod. Po prostu wprowadź go w oknie edycji. Pamiętaj, aby zapisać kod jako plik odd.py, zanim przejdziesz dalej.

I... *co teraz?* Jeśli masz podobnie jak my, nie możesz się już doczekać, aby uruchomić ten kod, prawda? Zróbmy to natychmiast. Korzystając z okna edycji, w którym wprowadzony został kod (jak widać powyżej), naciśnij klawisz *F5*. Gdy to zrobisz, może zdarzyć się kilka rzeczy...

Co się dzieje dalej...

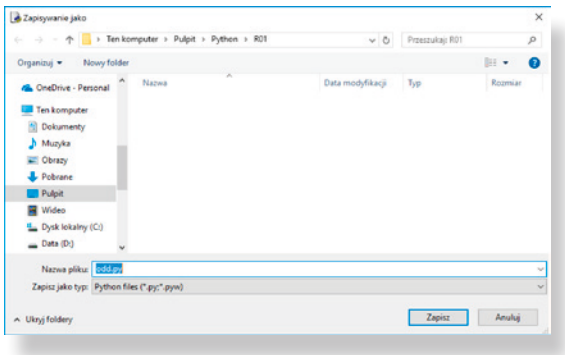
Jeśli uda Ci się uruchomić kod bez wystąpienia błędu, przejdź od razu do następnej strony i *kontynuuj lekturę*.

Jeśli zapomniałeś zapisać kod *przed* próbą jego uruchomienia, IDLE poskarży się, że *najpierw* powinieneś zapisać zmiany w pliku. Na ekranie ujrzysz w takiej sytuacji okno dialogowe podobne do tego, które widać poniżej.



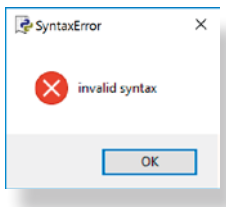
Standardowo IDLE nie uruchomi kodu, który nie został zapisany.

Kliknij przycisk *OK*, następnie podaj nazwę swojego pliku. Wybraliśmy dla niego nazwę *odd* i dodaliśmy rozszerzenie *py* (zgodnie z przyjętą dla Pythona konwencją, której warto przestrzegać).



Program możesz nazwać dowolnie, ale — jeśli podążasz za nami — najlepiej chyba będzie, jeśli użyjesz tej samej nazwy co my.

Jeśli kod uda się teraz (tj. po zapisaniu) uruchomić, przejdź do następnej strony i *kontynuuj lekturę*. Jeśli jednak pojawił się w nim jakiś błąd składni, na ekranie ujrzysz okno z komunikatem przedstawionym poniżej.



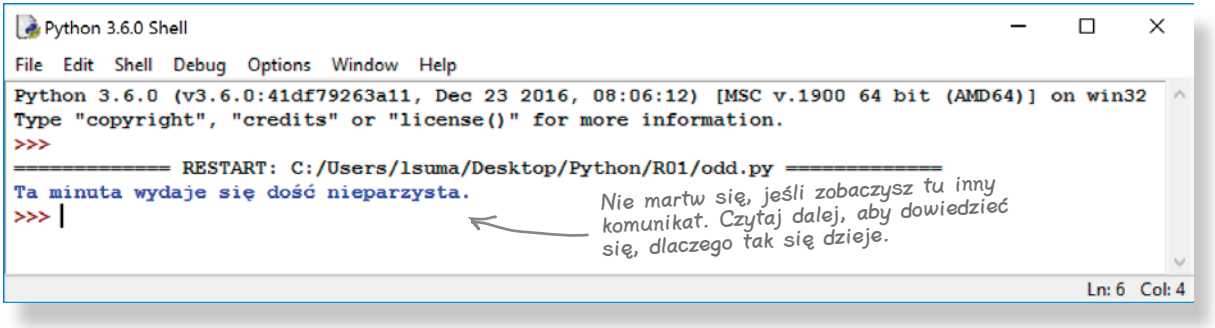
Jak już pewnie zauważyłeś, środowisko IDLE nie jest zbyt dobre w orzekaniu, z jakim błędem składni ma do czynienia. Kliknij jednak przycisk OK, a IDLE podświetli na czerwono miejsce w kodzie, w którym prawdopodobnie znajduje się błąd.

Kliknij przycisk *OK*, a następnie spójrz w miejsce, w którym według IDLE znajduje się błąd składni: poszukaj w tym celu czerwonego prostokąta w oknie edycji. Upewnij się, że Twój kod dokładnie odpowiada naszemu, jeszcze raz zapisz plik, po czym naciśnij klawisz *F5*, aby poprosić IDLE o ponowne wykonanie kodu.

Naciśnij klawisz F5, aby uruchomić kod

Naciśnięcie klawisza *F5* powoduje wykonanie kodu znajdującego się w obecnie wybranym oknie edycji środowiska IDLE, oczywiście przy założeniu, że kod ten nie zawiera żadnego błędu czasu wykonania. Jeśli błąd taki występuje, ujrzysz komunikat błędu **Traceback** (wyświetlany na czerwono). Przeczytaj ten komunikat i wróć do okna edycji, aby upewnić się, że kod, który wprowadziłeś, jest dokładnie taki sam jak ten, który Ci zaprezentowaliśmy. Zapisz poprawiony kod, po czym ponownie naciśnij klawisz *F5*. Gdy my to zrobiliśmy, powłoka Pythona stała się aktywnym oknem, a w środku zobaczyliśmy to, co widać na przedstawionym rysunku.

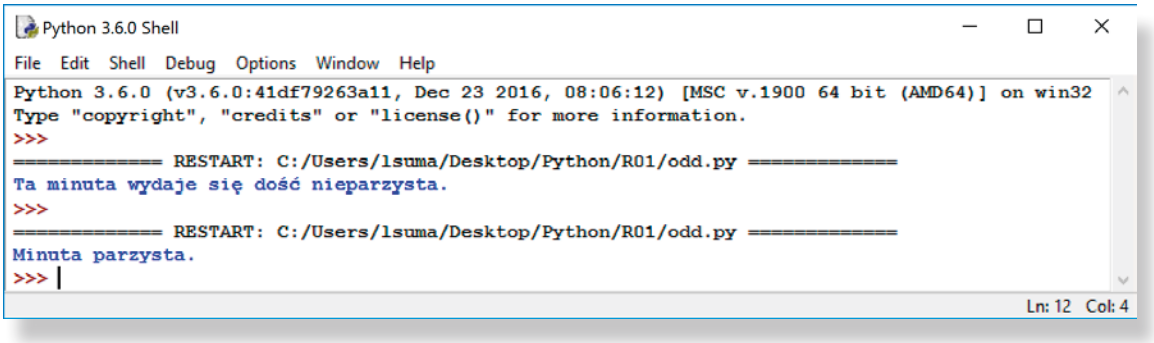
Odtąd „okno edycji środowiska IDLE" będziemy nazywać po prostu „oknem edycji".



Nie martw się, jeśli zobaczysz tu inny komunikat. Czytaj dalej, aby dowiedzieć się, dlaczego tak się dzieje.

Zależnie od tego, która była godzina, zamiast powyższego mogłeś ujrzeć komunikat *Minuta parzysta*. Nie martw się, jeśli tak się właśnie stało, ponieważ program wyświetla jeden z tych dwóch tekstów w zależności od tego, czy wartość minut bieżącego czasu Twojego komputera jest liczbą nieparzystą czy parzystą (uprzedzaliśmy w końcu, że przykład jest *wydumany*, prawda?). Jeśli poczekasz minutę, a następnie przejdziesz do okna edycji i ponownie naciśniesz klawisz *F5*, kod zostanie uruchomiony po raz kolejny. Tym razem zobaczysz inny komunikat (przy założeniu, że poczekałeś wymaganą minutę). Możesz śmiało uruchamiać ten kod tak często, jak tylko masz ochotę. Oto co ujrzeliśmy, gdy (bardzo cierpliwie) odczekaliśmy niezbędną minutę.

Naciśnięcie klawisza F5 w oknie edycji powoduje uruchomienie kodu, a następnie wyświetlenie będących wynikiem jego działania danych wyjściowych w oknie Python Shell.



Poświęćmy trochę czasu na zrozumienie działania tego kodu.

Kod zostaje uruchomiony natychmiast

Gdy IDLE prosi Pythona o uruchomienie kodu znajdującego się w oknie edycji, rozpoczyna on swoją pracę od początku pliku i od razu zaczyna wykonywać kod.

Dla tych, którzy korzystali wcześniej z języków programowania w rodzaju C, mamy informację: w Pythonie nie ma czegoś takiego jak metoda czy funkcja `main()`. Nie ma też znanego stamtąd procesu składającego się z edycji, kompilacji, łączenia i uruchamiania kodu. W przypadku Pythona po prostu edytuje się kod, zapisuje go i uruchamia *natychmiast*.

Moment, moment! Stwierdziliście, że „IDLE prosi Pythona o uruchomienie kodu...", ale czy Python nie jest językiem programowania, a IDLE środowiskiem programistycznym? Jeśli tak, co właściwie jest tu odpowiedzialne za uruchomienie?!?

A tak, złapałeś nas. To dość mylące.

Oto co powinieneś wiedzieć: „Python" jest nazwą nadaną językowi programowania, a „IDLE" to nazwa, którą otrzymało wbudowane IDE Pythona.

Oznacza to, że gdy instalujesz na swoim komputerze Pythona 3, instalowany jest wraz z nim **interpreter**, czyli program odpowiedzialny za uruchamianie kodu napisanego w języku Python. Dość mylące jest to, że interpreter ten jest również znany pod nazwą „Python". Poprawnie byłoby, gdyby każdy, odnosząc się do tego programu, używał bardziej prawidłowej nazwy, a więc nazywał go „interpreterem Pythona". Ale, niestety, nikt tego nie robi.

Począwszy od tej chwili w książce tej będziemy używać słowa „Python", mając na myśli język, a słowa „interpreter" — odnosząc się do narzędzia, które jest odpowiedzialne za uruchamianie kodu Pythona. Słowo „IDLE" określa IDE, które bierze Twój kod i uruchamia go za pomocą interpretera. To właśnie interpreter wykonuje tu całą rzeczywistą pracę.

Nie istnieją głupie pytania

P: Czy interpreter Pythona to coś w rodzaju wirtualnej maszyny Javy?

O: I tak, i nie. Tak, w tym sensie, że interpreter jest odpowiedzialny za uruchamianie kodu. Ale nie, jeśli chodzi o sposób, w jaki to robi. W przypadku Pythona nie istnieje coś takiego jak kompilacja kodu do postaci „wykonywalnej". W przeciwieństwie do Java VM interpreter nie uruchamia plików o rozszerzeniu `class`, po prostu uruchamia sam kod.

P: Ale kompilacja z pewnością musi się odbyć na jakimś etapie?

O: Tak, odbywa się, ale interpreter nie ujawnia tego procesu programiście korzystającemu z języka Python (czyli Tobie). Dba o wszystkie szczegóły i nie musisz się nimi przejmować. Ty widzisz jedynie to, że Twój kod jest uruchamiany, podczas gdy IDLE odwala całą brudną robotę, współpracując z interpreterem w Twoim imieniu. W dalszej części tej książki przedstawimy Ci więcej szczegółów na ten temat.

Uruchamianie kodu instrukcja po instrukcji

Poniżej znajdziesz kod programu zaprezentowanego na stronie 4.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

Zostańmy interpreterem języka Python

Poświęćmy trochę czasu, aby przejść przez ten kod w mniej więcej ten sam sposób, w jaki robi to interpreter — wiersz po wierszu, od *początku* pliku aż do jego *końca*.

W pierwszym wierszu kodu **importowane** są pewne istniejące funkcje z **biblioteki standardowej** Pythona, która stanowi ogromny zbiór modułów programistycznych zapewniających mnóstwo wstępnie zbudowanego (i wysokiej jakości) kodu wielokrotnego użytku.

Traktuj moduły jako kolekcje związanych ze sobą funkcji.

W naszym kodzie żądamy konkretnie jednego submodułu należącego do modułu `datetime`, który wchodzi w skład biblioteki standardowej. Nieco mylące jest to, że submoduł ten również nosi nazwę `datetime`, ale tak to już jest. Submoduł `datetime` zapewnia mechanizm umożliwiający posługiwanie się czasem, o czym przekonasz się podczas lektury kilku kolejnych stron.

To jest nazwa submodułu

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]
...
```

To jest nazwa modułu biblioteki standardowej, z którego ma zostać zaimportowany kod wielokrotnego użytku.

Pamiętaj: interpreter zaczyna pracę od początku pliku i kontynuuje ją aż do jego końca, wykonując każdy wiersz kodu Pythona po kolei.

Gdy w tej książce chcemy, abyś zwrócił szczególną uwagę na jakiś wiersz kodu, zaznaczamy go za pomocą pogrubionej czcionki i specjalnego tła (tak jak zrobiliśmy to w tym przykładzie).

Funkcje + moduły = biblioteka standardowa

Biblioteka standardowa Pythona jest bardzo *bogata* i dostarcza mnóstwa kodu wielokrotnego użytku.

Przyjrzyjmy się innemu modułowi, o nazwie `os`, który zapewnia niezależny od platformy sposób interakcji z systemem operacyjnym kontrolującym pracę używanego przez Ciebie komputera (do modułu `datetime` wrócimy za chwilę). Skupmy się na jednej tylko z oferowanych przezeń funkcji, `getcwd`, która — gdy się ją wywoła — zwraca ścieżkę dostępu do *bieżącego katalogu roboczego*.

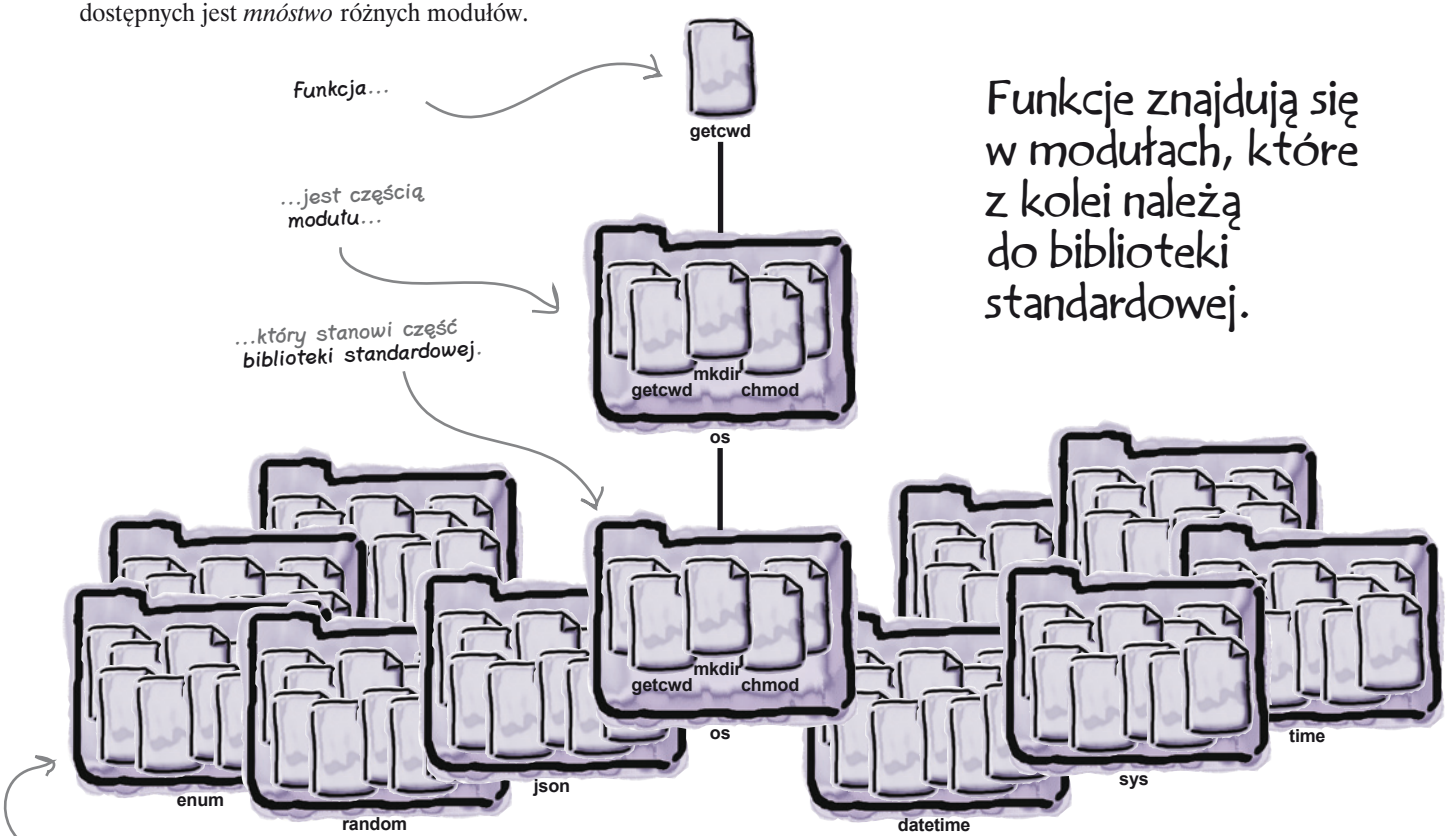
Oto typowy sposób *importowania*, a następnie *wywoływania* tej funkcji w programie napisanym w języku Python:

Import funkcji z odpowiedniego modułu...

```
from os import getcwd

where_am_I = getcwd()
```

...i wywołanie jej we właściwy sposób.

Zbiór związanych ze sobą funkcji tworzy moduł, a w bibliotece standardowej dostępnych jest *mnóstwo* różnych modułów.



Funkcje znajdują się w modułach, które z kolei należą do biblioteki standardowej.

Na tym etapie nie przejmuj się tym, do czego służy każdy z tych modułów. Z niektórymi z nich zapoznasz się pobieżnie na kolejnych stronach, a większość pozostałych zobaczysz jeszcze w dalszej części tej książki.

Biblioteka standardowa z bliska

Biblioteka standardowa stanowi klejnot w koronie języka Python, dostarczając moduły wielokrotnego użytku, które pomagają dosłownie we wszystkim, począwszy od na przykład posługiwania się danymi, poprzez manipulowanie archiwami ZIP, a skończywszy na wysyłaniu wiadomości poczty elektronicznej i działaniach związanych z HTML-em. Biblioteka standardowa zawiera nawet serwer WWW oraz obsługuje technologię bazodanową *SQLite*. W tej ramce zamieściliśmy przegląd zaledwie kilku spośród najczęściej używanych modułów biblioteki standardowej. Zamieszczone poniżej przykłady możesz uruchomić w oknie IDLE, wpisując je po znaku zachęty (>>>). Jeśli patrzysz w tej chwili na okno edycji środowiska, skorzystaj z polecenia menu *Run/Python Shell* (uruchom/powłoka Pythona), aby przejść do wiersza poleceń powłoki.

Zacznijmy od tego, by się dowiedzieć czegoś na temat systemu operacyjnego, w którym działa Twój interpreter. Choć Python szczyci się swoją wieloplatformowością, dzięki której kod napisany na jednej platformie może być uruchamiany (bez większych zmian) na innej, czasami zdarzają się sytuacje, gdy ważna jest wiedza na temat tego, jaki system operacyjny jest w użyciu. Pozyskanie tej wiedzy umożliwia moduł `sys`. Poniżej znajdziesz przykład przedstawiający sposób określenia rodzaju systemu, na którym działa interpreter. W tym celu należy najpierw zaimportować moduł `sys`, a następnie odwołać się do atrybutu `platform`.

```
>>> import sys
>>> sys.platform
'win32'
```

Zaimportuj moduł, którego potrzebujesz, po czym odwołaj się do atrybutu, który Cię interesuje. Wygląda na to, że korzystamy z „win32", co oznacza, że naszym systemem operacyjnym jest Windows.

Moduł `sys` stanowi dobry przykład modułu wielokrotnego użytku, który zapewnia przede wszystkim dostęp do atrybutów ustawień (takich jak `platform`). Innym przykładem może być sposób sprawdzenia wersji Pythona będącej w użyciu. Wynik zostanie przekazany funkcji `print` w celu wyświetlenia go na ekranie.

```
>>> print(sys.version)
3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12)
[MSC v.1900 64 bit (AMD64)]
```

Widać tu mnóstwo informacji na temat wersji Pythona, z której korzystamy, włącznie z tym, że jest to wersja oznaczona numerem 3.6.0.

Moduł `os` to z kolei dobry przykład modułu wielokrotnego użytku, który przede wszystkim oferuje rozmaite funkcje, jak również zapewnia niezależny od systemu operacyjnego sposób, w jaki kod Pythona może wchodzić w interakcję z systemem, w którym pracuje.

Poniżej znajdziesz na przykład fragment umożliwiający sprawdzenie nazwy folderu, w którym operuje Twój kod. Sposób ten wymaga zastosowania funkcji `getcwd`. Podobnie jak ma to miejsce w przypadku każdego innego modułu, również ten należy zaimportować przed wywołaniem funkcji.

```
>>> import os
>>> os.getcwd()
'C:\\Users\\RuszGlowa\\PrzykladoweKody'
```

Zaimportuj odpowiedni moduł i wywołaj funkcję, której potrzebujesz.

Do zmiennych środowiskowych systemu możesz uzyskać dostęp jako do całości (korzystając z atrybutu `environ`) lub indywidualnie (używając funkcji `getenv`).

```
>>> os.environ
'environ({'ALLUSERSPROFILE': 'C:\\ProgramData', 'APPDATA': 'C:\\Users\\RuszGlowa\\AppData\\Roaming',
'ASL.LOG': 'Destination=file', ..., 'PSMODULEPATH': 'C:\\WINDOWS\\system32\\WindowsPowerShell\\v1.0\\
Modules\\', ..., 'TMP': 'C:\\Users\\RuszGlowa\\AppData\\Local\\Temp', ..., 'USERNAME': 'RuszGlowa',
'USERPROFILE': 'C:\\Users\\RuszGlowa', 'WINDIR': 'C:\\WINDOWS'})'
>>> os.getenv('HOME')
'C:\\Users\\RuszGlowa'
```

Atrybut environ zawiera mnóstwo danych.

Za pomocą funkcji getenv możesz uzyskać dostęp do atrybutu o określonej nazwie (należącego do danych zawartych w atrybucie environ).

Biblioteka standardowa z bliska — ciąg dalszy

Operacje na datach (i godzinach) są bardzo powszechne, a biblioteka standardowa zapewnia moduł `datetime`, który ułatwia przeprowadzanie działań na danych tego rodzaju. Funkcja `date.today` umożliwia sprawdzenie aktualnej daty, tak jak zostało to pokazane poniżej.

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2017, 5, 27)
```

Dzisiejsza data

To jednak z pewnością dość dziwny sposób wyświetlania bieżącej daty, nieprawdaż? Można odwołać się z osobna do każdej wartości reprezentującej datę (tj. sprawdzić osobno dzień, miesiąc i rok), dodając do wywołania funkcji `date.today` odniesienie do odpowiedniego atrybutu.

```
>>> datetime.date.today().day
27
>>> datetime.date.today().month
5
>>> datetime.date.today().year
2017
```

Części składowe dzisiejszej daty

Możesz również wywołać funkcję `date.isoformat`, przekazując jej dzisiejszą datę, aby wyświetlić ją w formacie znacznie przyjaźniejszym dla użytkownika.

```
>>> datetime.date.isoformat(datetime.date.today())
'2017-05-27'
```

Dzisiejsza data w formacie tekstowym

Jest również czas, którego nikt z nas wydaje się nie mieć w nadmiarze. Czy *biblioteka standardowa* może nas poinformować, która godzina jest teraz? Tak. Wystarczy, że po zaimportowaniu modułu `time` wywołasz funkcję `strftime` i określisz, jak wyświetlony ma być bieżący czas. W tym przypadku jesteśmy zainteresowani bieżącą godziną (`%H`) i minutą (`%M`) w formacie 24-godzinnym.

```
>>> import time
>>> time.strftime("%H:%M")
'23:55'
```

Wielkie nieba! Naprawdę jest już tak późno?

A co, jeśli chcesz sprawdzić dzień tygodnia i to, czy jest jeszcze przed południem, czy już po południu? Możesz to zrobić, korzystając z funkcji `strftime` oraz odpowiednich specyfikatorów.

```
>>> time.strftime("%A %p")
'Sunday PM'
```

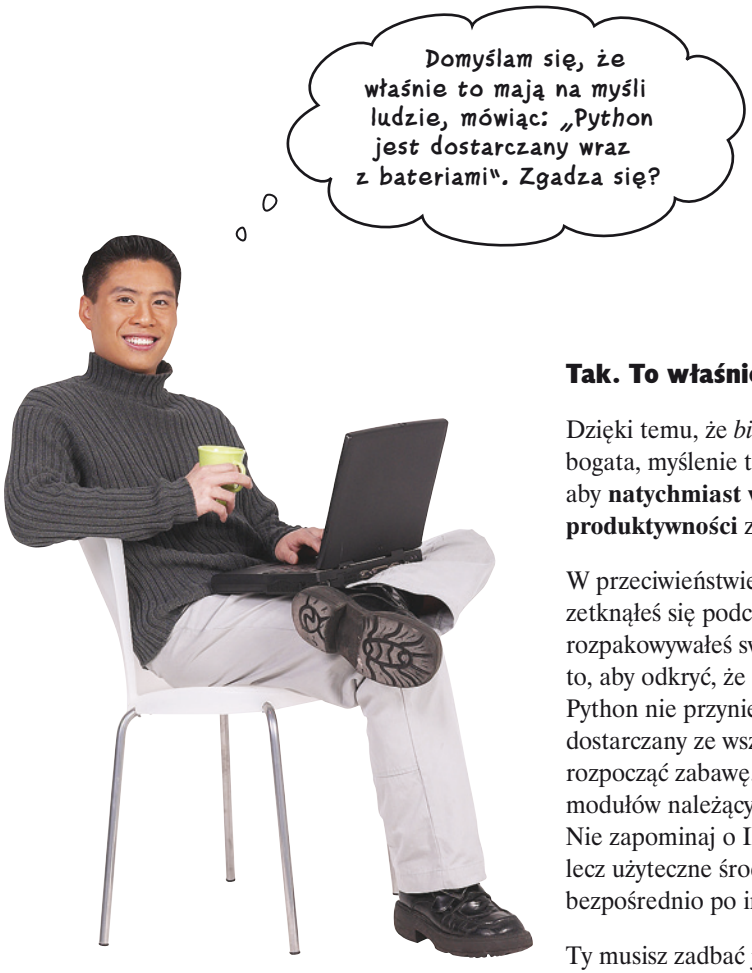
Dowiedzieliśmy się, że zostało pięć minut do północy w niedzielny wieczór... Może już czas do łóżka?

Pora na ostatni przykład typu funkcji wielokrotnego użytku zapewnianych przez *bibliotekę standardową*. Wyobraź sobie, że masz pewien dokument HTML, w którym — jak podejrzewasz — mogą występować potencjalnie niebezpieczne znaczniki `<script>`. Zamiast parsować rzeczony dokument w celu wykrycia i usunięcia tych znaczników, może warto zakodować wszystkie nawiasy trójkątne sprawiające problemy za pomocą należącej do modułu `html` funkcji `escape`. A może masz już jakiś zakodowany plik HTML, któremu chciałbyś przywrócić pierwotną postać? Pomoże Ci w tym funkcja `unescape`. Poniżej znajdziesz przykład zastosowania obydwu wspomnianych funkcji.

```
>>> import html
>>> html.escape("Ten fragment dokumentu HTML zawiera znacznik <script>skryptu</script>.")
'Ten fragment dokumentu HTML zawiera znacznik &lt;script&gt;skryptu&lt;/script&gt;.'
>>> html.unescape("Kocham &lt;bibliotekę standardową&gt; Pythona.")
'Kocham <bibliotekę standardową> Pythona.'
```

Konwersja na postać zakodowanego tekstu HTML i z powrotem

Baterie w zestawie



Tak. To właśnie mają na myśli.

Dzięki temu, że *biblioteka standardowa* jest *tak* bogata, myślenie to jedyne, czego potrzebujesz, aby **natychmiast wspiąć się na wyżyny produktywności** z Pythonem na pokładzie.

W przeciwieństwie do sytuacji, z którą nieraz zetknąłeś się podczas wigilijnego wieczoru, gdy rozpakowywałeś swoją nową zabawkę tylko po to, aby odkryć, że nie dołączono do niej baterii, Python nie przyniesie Ci rozczarowania, jest bowiem dostarczany ze wszystkim, co potrzebne, aby rozpocząć zabawę. Prawda ta nie dotyczy wyłącznie modułów należących do *biblioteki standardowej*. Nie zapominaj o IDLE, zapewniającym niewielkie, lecz użyteczne środowisko programistyczne już bezpośrednio po instalacji Pythona.

Ty musisz zadbać jedynie o kod.

Nie istnieją głupie pytania

P: Skąd niby mam wiedzieć, jakie możliwości zapewniają poszczególne moduły należące do biblioteki standardowej?

O: Dokumentacja Pythona zawiera odpowiedzi na wszystkie pytania dotyczące biblioteki standardowej. Oto właściwy punkt startowy: *https://docs.python.org/3/library/index.html*.

Dla maniaków

Biblioteka standardowa to niejedyne miejsce, gdzie znajdziesz doskonałe, gotowe do zaimportowania moduły, które możesz wykorzystać w swoim kodzie. Społeczność programistów Pythona buduje stale powiększającą się kolekcję modułów pochodzących od osób trzecich. Niektóre z tych modułów przedstawimy bliżej w dalszej części tej książki. Jeśli chcesz zerknąć na tę kolekcję, sprawdź utrzymywane przez społeczność repozytorium, które znajduje się pod adresem *http://pypi.python.org*.

Struktury danych są wbudowane

Poza znakomitą *biblioteką standardową* Python zapewnia również potężne wbudowane **struktury danych**. Jedną z nich jest **lista**, którą można uważać za *tablicę* na sterydach. Listy, podobnie jak znane z innych języków programowania tablice, umieszcza się w Pythonie w nawiasach kwadratowych ([]).

Kolejne trzy wiersze kodu naszego programu (pokazane poniżej) przypisują *literalną* listę liczb nieparzystych do zmiennej o nazwie odds. W tym fragmencie kodu odds jest *listą liczb całkowitych*, ale listy w języku Python mogą zawierać *dowolne* dane *jakiegokolwiek* typu, a ponadto w ramach jednej listy możesz nawet zastosować różne typy danych (jeśli tylko masz na to ochotę). Zwróć uwagę na fakt, że lista odds rozciąga się na trzy wiersze, mimo że stanowi pojedynczą instrukcję. To jak najbardziej w porządku, ponieważ interpreter nie stwierdzi, że instrukcja została zakończona, aż do momentu znalezienia zamykającego nawiasu kwadratowego (]), który będzie pasował do występującego wcześniej nawiasu otwierającego ([). **W Pythonie koniec wiersza oznacza zwykle koniec instrukcji**, zdarzają się jednak wyjątki od tej ogólnej reguły, a wielowierszowe listy są tylko jednym z nich (z innymi zapoznasz się w dalszej części książki).

Podobnie jak tablice, listy mogą przechowywać dane dowolnego typu.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]
                ...
```

To nowa zmienna o nazwie „odds"; jest ona przypisana do listy liczb nieparzystych.

To jest lista liczb nieparzystych, umieszczona w nawiasach kwadratowych. Ta jedna instrukcja rozciąga się na trzy wiersze, co nie stanowi najmniejszego problemu.

Z listami można robić wiele różnych rzeczy, jednak na dalsze informacje na ten temat będziesz musiał poczekać do czasu rozpoczęcia lektury jednego z kolejnych rozdziałów tej książki. Wszystko, co musisz teraz wiedzieć, to to, że nasza lista *istnieje*, została *przypisana* do zmiennej odds (dzięki zastosowaniu **operatora przypisania**, czyli =) oraz *zawiera* przedstawione powyżej liczby.

Zmienne Pythona są przypisywane dynamicznie

Przed przejściem do kolejnego wiersza kodu przyda się z pewnością kilka słów na temat zmiennych, zwłaszcza jeśli należysz do tych programistów, którzy przyzwyczaili się do zmiennych wstępnie deklarowanych wraz z informacją o typie jeszcze *przed* pierwszym użyciem ich w kodzie (jak ma to miejsce w przypadku statycznie typowanych języków programowania).

Python oferuje wszystkie standardowe operatory, w tym <, >, <=, >=, ==, !=, jak również operator przypisania =.

W Pythonie zmienne zaczynają istnieć w momencie ich pierwszego użycia, a **ich typ nie musi być wstępnie deklarowany**. Zmienne Pythona pozyskują informację o swoim typie od obiektu, który jest do nich przypisany. W naszym programie zmiennej odds przypisana zostaje lista liczb, dlatego odds jest w tym przypadku listą.

Przyjrzyjmy się innej instrukcji przypisania zmiennej. Tak się szczęśliwie złożyło, że instrukcja taka znajduje się akurat w następnym wierszu kodu naszego programu.

Wywoływanie metod daje wyniki

Trzeci wiersz naszego kodu zawiera kolejną **instrukcję przypisania**.

W przeciwieństwie do poprzedniego nie następuje tu przypisanie struktury danych do zmiennej, lecz przypisanie **wyniku** wywołania metody do kolejnej nowej zmiennej, noszącej nazwę `right_this_minute`. Raz jeszcze przyjrzyj się trzeciemu wierszowi kodu.

Tutaj znajduje się kolejna zmienna, która zostaje utworzona i otrzymuje wartość.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

To wywołanie skutkuje wygenerowaniem wartości, która zostaje przypisana do zmiennej.

Wywoływanie funkcji modułów wbudowanych

W trzecim wierszu kodu wywoływana jest metoda o nazwie `today` wchodząca w skład submodułu `datetime`, który *sam* z kolei jest częścią składową modułu `datetime` (wspominaliśmy już, że zastosowana tu strategia nazewnictwa *była* nieco myląca). To, że funkcja `today` jest wywoływana, możesz stwierdzić dzięki występowaniu po jej nazwie standardowych nawiasów `()`.

Wywołanie funkcji `today` skutkuje zwróceniem „obiektu czasu”, który zawiera wiele cząstek informacji na temat bieżącego czasu. Stanowią one **atrybuty** czasu, do których możesz się dostać, korzystając ze zwyczajowej składni **notacji kropkowej**. W naszym programie interesuje nas atrybut minut, do którego dostęp możemy uzyskać, dodając na końcu wywołania funkcji ciąg znaków `.minute`, co pokazano powyżej. Wartość będąca wynikiem zostaje następnie przypisana do zmiennej `right_this_minute`. Możesz przyjąć, że znaczenie tego wiersza kodu jest następujące: *utwórz obiekt reprezentujący bieżący czas, a następnie wyodrębnij wartość atrybutu minut przed przypisaniem jej do zmiennej*. Kusząca wydaje się perspektywa *podzielenia* tego wiersza kodu na dwa w celu „ułatwienia jego zrozumienia”, co zostało zaprezentowane poniżej.

W dalszej części książki zobaczysz więcej składni wykorzystującej notację kropkową.

Najpierw sprawdź bieżący czas...

```
time_now = datetime.today()
right_this_minute = time_now.minute
```

...następnie wyodrębnij wartość minut.

Możesz to zrobić (jeśli tylko masz ochotę), ale większość programistów Pythona woli **nie** tworzyć tymczasowych zmiennych (`time_now` w tym przypadku), *chyba że* jest to naprawdę potrzebne w jakimś miejscu programu.

Decydowanie, kiedy należy wykonać blok kodu

Na tym etapie mamy już listę liczb o nazwie `odds`. Dysponujemy też zmienną o nazwie `right_this_minute` przechowującą minutę. Aby dowiedzieć się, czy wartość bieżącej minuty zapisana w zmiennej `right_this_minute` jest liczbą parzystą, czy też nieparzystą, potrzebujemy jakiegoś sposobu, który umożliwi nam sprawdzenie, czy znajduje się ona na liście `odds`. Ale jak mamy to zrobić?

Okazuje się, że język Python bardzo upraszcza tego typu operacje. Poza tym, że oferuje wszystkie typowe operatory porównania, które spodziewasz się znaleźć w dowolnym języku programowania (takie jak >, <, >= czy <=), zapewnia również kilka własnych „superoperatorów", pośród których jest operator `in`.

Operator `in` umożliwia sprawdzenie, czy jedna rzecz znajduje się *wewnątrz* innej. Przyjrzyj się następnemu wierszowi kodu naszego programu, w którym to wierszu operator `in` jest wykorzystywany do sprawdzenia, czy wartość przechowywana w zmiennej `right_this_minute` znajduje się na liście `odds`.

```
...
right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("Ta minuta wydaje się dość nieparzysta.")
...
```

W wyniku wykonania tej instrukcji if otrzymamy wartość True lub False.

Operator in ma potężne możliwości. Za jego pomocą można stwierdzić, czy jedna rzecz znajduje się wewnątrz innej.

Operator `in` zwraca wartość `True` lub `False`. Jak z pewnością się spodziewasz, jeśli wartość przechowywana w zmiennej `right_this_minute` znajduje się na liście `odds`, w wyniku wykonania instrukcji `if` otrzymamy wartość `True`, a blok kodu związany z tą instrukcją `if` zostanie wykonany.

Bloki w języku Python łatwo jest zauważyć, ponieważ są zawsze odpowiednio wcięte.

W naszym programie znajdują się dwa bloki kodu, z których każdy zawiera pojedyncze wywołanie funkcji `print`. Funkcja ta umożliwia wyświetlanie komunikatów na ekranie (i w tej książce będziemy się nią posługiwać naprawdę często). Gdy wprowadzisz kod tego programu do okna edycji, być może zauważysz, że IDLE ułatwia Ci zadanie, automatycznie dodając odpowiednie wcięcia. To bardzo wygodne, ale zawsze musisz sprawdzać, czy wcięcia, które zrobiło IDLE, na pewno są zgodne z Twoimi intencjami.

```
...
right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("Ta minuta wydaje się dość nieparzysta.")
else:
    print("Minuta parzysta.")
...
```

Tutaj znajduje się jeden blok kodu. Zauważ, że jest on wcięty.

Funkcja print umożliwia wyświetlenie komunikatu na standardowym urządzeniu wyjściowym (na przykład na ekranie Twojego komputera).

A tutaj znajduje się kolejny blok kodu. Zauważ, że również on jest wcięty.

Czy zauważyłeś, że nie ma tu nigdzie nawiasów klamrowych?

Co się stało z moimi nawiasami klamrowymi?

Jeśli przywykłeś do języka programowania, w którym używa się nawiasów klamrowych ({ oraz }) do wyznaczania granic bloków kodu, to pierwszy kontakt z blokami w języku Python może być dość dezorientujący, jako że w Pythonie sprawy mają się w tej kwestii zupełnie inaczej. Zamiast nawiasów klamrowych do oznaczania bloków kodu korzysta się tu wyłącznie z **wcięć**, a sam *blok* programiści Pythona wolą dla odmiany określać mianem **zestawu** (ang *suite*), tak żeby było ciekawiej.

To nie tak, że nawiasy klamrowe nie są używane w języku Python. Jak najbardziej są, ale — o czym przekonasz się, czytając rozdział 3. — mają tu większy związek z wyznaczaniem granic danych niż zestawów (czyli *bloków*) kodu.

Zestawy łatwo zauważyć w każdym programie napisanym w języku Python, ponieważ są zawsze wcięte. Ułatwia to naszym mózgom szybką identyfikację zestawu podczas czytania kodu. Inną podpowiedzią graficzną, za którą warto się tu rozglądać, jest znak dwukropka (`:`) — jest on wykorzystywany do wskazywania zestawu związanego z którąś z instrukcji sterujących Pythona (taką jak `if`, `else` czy `for`). Z przykładami zastosowania tych elementów będziesz się spotykał właściwie w całej tej książce.

Zamiast mówić o „blokach" kodu, programiści korzystający z języka Python używają słowa „zestaw". W praktyce stosowane są obydwa te terminy, ale w dokumentacji Pythona korzysta się raczej z tego drugiego.

Dwukropek poprzedza wcięty zestaw kodu

Dwukropek (`:`) jest istotny, ponieważ poprzedza nowy zestaw kodu, który musi być wcięty w prawo w stosunku do poprzedzającej go instrukcji. Jeśli zapomnisz o dodaniu wcięcia po znaku dwukropka, interpreter zgłosi błąd przy próbie uruchomienia kodu.

W naszym przykładzie nie tylko instrukcja `if` zakończona jest znakiem dwukropka, tak samo jest z instrukcją `else`. Poniżej raz jeszcze zaprezentowaliśmy cały kod naszego przykładowego programu.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

Dwukropki poprzedzają wcięte zestawy.

Dotarliśmy niemal do końca. Została nam już do omówienia tylko jedna instrukcja.

Co jeszcze może Ci dać instrukcja warunkowa?

Doszliśmy już niemal do końca omawiania kodu naszego przykładowego programu, w którym do wyjaśnienia pozostał nam tylko jeden wiersz. Nie jest to wielki wiersz, ale ma dość duże znaczenie: chodzi o instrukcję `else`, która wskazuje blok kodu wykonywany, jeśli okaże się, że warunek sprawdzany przez odpowiednią instrukcję `if` ma wartość `False`.

Przyjrzyj się bliżej instrukcji `else` występującej w naszym przykładowym kodzie. Jej wcięcie musimy usunąć, aby zrównać ją z częścią `if` wyrażenia warunkowego.

```
if right_this_minute in odds:
    print("Ta minuta wydaje się dość nieparzysta.")
else:
    print("Minuta parzysta.")
```

Czy widzisz dwukropek?

Czy zauważyłeś, że słowo „else" nie ma wcięcia i jest wyrównane ze słowem „if"?

Zapominanie o dwukropku to typowa pomyłka pythonowych nowicjuszy piszących kod po raz pierwszy.

Domyślam się, że skoro istnieje „else", musi być też jakieś „else if", a może w Pythonie jest to „elseif"?

Ani jedno, ani drugie. W Pythonie jest to `elif`.

Jeśli w ramach jednej instrukcji warunkowej musisz sprawdzić wiele warunków, możesz skorzystać z oferowanych przez język Python słów kluczowych `elif` oraz `else`. Możesz użyć tylu instrukcji `elif`, ilu potrzebujesz (a każda z nich będzie miała własny zestaw).

Oto niewielki przykład, w którym założyliśmy, że zmiennej o nazwie `today` został wcześniej przypisany łańcuch znakowy reprezentujący bieżący dzień tygodnia.

```
if today == 'Sobota':
    print('Impreza!!!')
elif today == 'Niedziela':
    print('Rekonwalescencja.')
else:
    print('Praca, praca, praca.')
```

Trzy niezależne zestawy: jeden dla instrukcji if, drugi dla elif i ostatni, obsługujący wszystkie pozostałe przypadki, dla else.

Zestawy mogą zawierać zagnieżdżone zestawy

Każdy zestaw może zawierać dowolną liczbę zagnieżdżonych zestawów, które również muszą być odpowiednio wcięte. Gdy programiści Pythona rozmawiają o zagnieżdżonych zestawach, zwykle mówią o **poziomach wcięcia**.

Początkowy poziom wcięcia każdego programu jest przeważnie określany mianem *pierwszego* lub (z racji tego, że właśnie tak liczy się w wielu językach programowania) *zerowego* poziomu wcięcia. Kolejne nazywa się drugim, trzecim, czwartym i tak dalej (lub też poziomem jeden, poziomem dwa, poziomem trzy i tak dalej).

Oto nieco bardziej rozbudowana wersja przykładu przedstawionego na poprzedniej stronie. Zauważ, w jaki sposób zagnieżdżona instrukcja `if/else` została dodana do instrukcji `if`, która zostaje wykonana, jeśli zmienna `today` ma wartość `'Niedziela'`. Zakładamy tu również, że inna zmienna, o nazwie `condition`, istnieje i ma wartość określającą Twoje aktualne samopoczucie. Zaznaczyliśmy, gdzie znajduje się każdy z zestawów oraz na którym poziomie wcięcia występuje.

```
if today == 'Sobota':
    print('Impreza!!!')
elif today == 'Niedziela':
    if condition == 'Ból głowy':
        print('Rekonwalescencja, potem odpoczynek.')
    else:
        print('Odpoczynek.')
else:
    print('Praca, praca, praca.')
```

Ten pojedynczy wiersz kodu jest zestawem.

Obydwa te pojedyncze wiersze kodu są zestawami.

Te cztery wiersze kodu stanowią zestaw.

Ten pojedynczy wiersz kodu jest zestawem.

Wcięcie poziomu zero

Wcięcie poziomu jeden

Wcięcie poziomu dwa

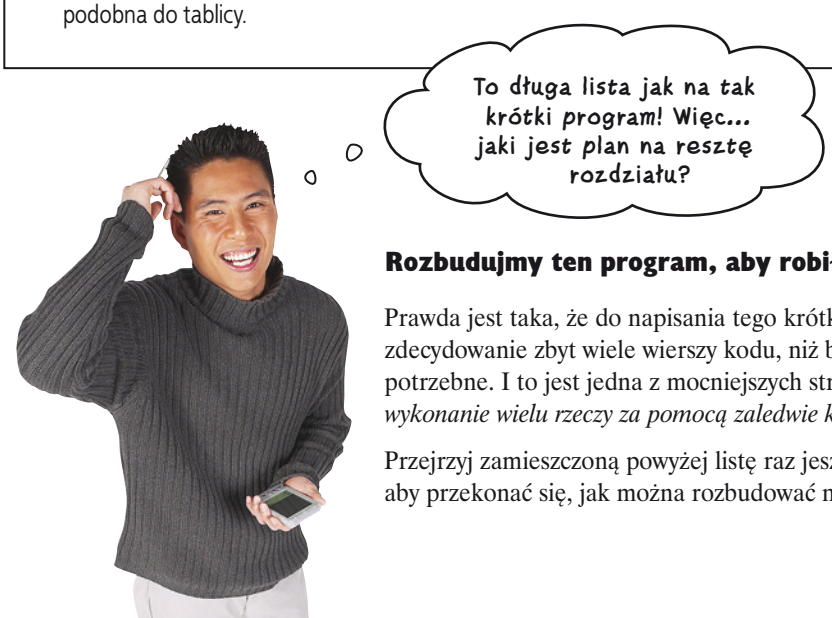
Warto tu odnotować, że kod znajdujący się na pewnym poziomie wcięcia jest związany z kodem znajdującym się na tym samym poziomie tylko wtedy, gdy cały ten kod występuje *w obrębie tego samego zestawu*. W przeciwnym razie należą one do oddzielnych zestawów i nie ma znaczenia, że współdzielą ten sam poziom wcięcia. Kluczowe znaczenie ma to, że wcięcia są wykorzystywane w języku Python do rozgraniczania zestawów kodu.

Co już wiemy

Po omówieniu kilku ostatnich wierszy kodu nadszedł czas na chwilę przerwy, którą poświęcimy podsumowaniu tego, co program `odd.py` powiedział nam na temat Pythona.

CELNE SPOSTRZEŻENIA

- Python dostarczany jest wraz z wbudowanym IDE o nazwie IDLE, które umożliwia tworzenie, edytowanie i uruchamianie kodu, stanowiąc kompletne środowisko do pisania programów, zapisywania ich na dysku, a następnie wykonywania za pomocą klawisza *F5*.
- IDLE współpracuje z interpreterem Pythona, który automatyzuje proces kompilacji, łączenia i uruchamiania programów. Pozwala to skoncentrować się na pisaniu kodu.
- Interpreter uruchamia kod (przechowywany w pliku) od początku do końca, po jednym wierszu naraz. W Pythonie nie ma czegoś takiego jak metoda lub funkcja `main()`.
- Python wyposażony jest w potężną bibliotekę standardową, która zapewnia dostęp do mnóstwa modułów wielokrotnego użytku (jednym z przykładów może tu być moduł `datetime`).
- W tworzonych w Pythonie programach można wykorzystywać standardowe struktury danych należące do zbioru oferowanego przez ten język. Jedną z tych struktur jest lista, która jest bardzo podobna do tablicy.
- Typ zmiennej nie musi być zadeklarowany. Gdy zmiennej przypisuje się w Pythonie wartość, zmienna ta dynamicznie przyjmuje typ przypisywanej wartości.
- Decyzje podejmuje się za pomocą instrukcji `if/elif/else`. Słowa kluczowe `if`, `elif` oraz `else` poprzedzają bloki kodu, które w świecie Pythona znane są pod nazwą „zestawy".
- Łatwo jest rozpoznać zestawy kodu, ponieważ zawsze są wcięte. Wcięcie stanowi jedyny mechanizm grupowania kodu zapewniany przez język Python.
- Zestawy kodu są nie tylko wcięte, lecz również poprzedzone znakiem dwukropka (`:`). Jest to syntaktyczny wymóg języka Python.



Rozbudujmy ten program, aby robił coś więcej.

Prawda jest taka, że do napisania tego krótkiego programu wykorzystaliśmy zdecydowanie zbyt wiele wierszy kodu, niż było to w rzeczywistości potrzebne. I to jest jedna z mocniejszych stron Pythona: *umożliwia on wykonanie wielu rzeczy za pomocą zaledwie kilku wierszy kodu*.

Przejrzyj zamieszczoną powyżej listę raz jeszcze i przejdź na następną stronę, aby przekonać się, jak można rozbudować nasz program.

Rozbudowa naszego programu, aby robił więcej

Rozszerzmy nasz program, aby nauczyć się nieco więcej o języku Python.

W tej chwili program wykonuje się raz, po czym przerywa swoje działanie. Wyobraź sobie, że chcielibyśmy, aby nasz program wykonywał się więcej razy; powiedzmy, pięciokrotnie. A dokładniej: wykonajmy „kod sprawdzania minuty" oraz instrukcję `if/else` pięć razy, wstrzymując jego działanie na losową liczbę sekund pomiędzy każdym wyświetleniem komunikatu (wyłącznie po to, aby było ciekawiej). Gdy program dobiegnie końca, na ekranie powinno być widocznych pięć komunikatów, a nie tylko jeden, jak miało to miejsce wcześniej.

Ponownie zamieściliśmy nasz kod, tym razem zakreślając w nim fragment, który chcemy wykonać wielokrotnie.

Zmieńmy nasz program tak, aby ten kod wykonywał się kilka razy.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

Co powinniśmy zrobić

1. **Zapętlić zakreślony kod.**
   Pętla umożliwia powtarzanie dowolnego zestawu, a język Python oferuje wiele sposobów przeprowadzenia tej operacji. W naszym przypadku do iteracji wykorzystamy pętlę `for` (bez analizowania powodów, dla których tak będzie).

2. **Wstrzymać wykonanie.**
   Standardowy moduł Pythona o nazwie `time` zapewnia funkcję `sleep`, która umożliwia wstrzymanie działania programu na określoną liczbę sekund.

3. **Wygenerować liczbę losową.**
   Tak się szczęśliwie składa, że inny moduł języka Python, `random`, zawiera funkcję o nazwie `randint`, której możemy użyć, aby wygenerować liczbę losową. Skorzystajmy z niej w celu otrzymania liczby z przedziału od 1 do 60, po czym użyjmy jej, aby wstrzymać wykonanie naszego programu w każdej iteracji.

Teraz wiemy już, co należy zrobić. Ale czy istnieje jakiś preferowany sposób wprowadzenia odpowiednich zmian?

Jak najlepiej poradzić sobie z tym problemem?

Wiesz, co masz robić: pochyl głowę, przeczytaj dokumentację i opracuj w Pythonie kod, którego potrzebujesz, aby poradzić sobie z tym problemem. Po zrobieniu tego będziesz gotów do wprowadzenia odpowiednich zmian w swoim programie...

To podejście działa, ale z natury jestem raczej eksperymentatorką. Lubię wypróbowywać niewielkie fragmenty kodu, zanim zdecyduję się na wprowadzenie zmian do swojego działającego programu. Cieszy mnie czytanie dokumentacji, ale lubię też eksperymentować...

W przypadku Pythona sprawdzą się obydwa podejścia

Pisząc w Pythonie, możesz skorzystać z *obydwu* tych sposobów, ale większość programistów posługujących się tym językiem woli **eksperymentować**, gdy próbuje opracować kod potrzebny w danej sytuacji.

Nie zrozum nas źle: bynajmniej nie sugerujemy, że podejście Roberta jest niewłaściwe, a Laury dobre. Chodzi tylko o to, że programiści Pythona mają do dyspozycji obydwie te opcje, a powłoka Pythona (z którą zetknąłeś się na krótko na początku niniejszego rozdziału) sprawia, że eksperymentowanie jest naturalnym wyborem dla osób korzystających z tego języka.

Wykonując eksperymenty w wierszu poleceń Pythona, sprawdźmy, jakiego kodu potrzebujemy, aby rozbudować nasz program.

Eksperymentowanie w wierszu poleceń (> > >) ułatwia opracowanie potrzebnego kodu.

Powrót do powłoki Pythona

Poniżej możesz zobaczyć, jak wyglądało okno *Python Shell* w chwili, gdy korzystaliśmy z niego po raz ostatni (Twoje okno może wyglądać nieco inaczej, jako że komunikaty mogły się w nim pokazać w odwrotnej kolejności).

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
============= RESTART: C:/Users/lsuma/Desktop/Python/R01/odd.py =============
Ta minuta wydaje się dość nieparzysta.
>>>
============= RESTART: C:/Users/lsuma/Desktop/Python/R01/odd.py =============
Minuta parzysta.
>>> |
                                                                  Ln: 12  Col: 4
```

Powłoka Pythona (albo po prostu krócej: „powłoka”) wyświetlała komunikaty naszego programu, ale może robić znacznie więcej. Wiersz poleceń umożliwia wprowadzenie po znaku zachęty >>> dowolnych instrukcji kodu Pythona i *natychmiastowe* ich wykonanie. Jeśli instrukcja generuje jakieś dane wyjściowe, powłoka je wyświetla. Jeśli wynikiem wykonania instrukcji jest jakaś wartość, powłoka wyświetli tę obliczoną wartość. Jeśli jednak tworzysz nową zmienną i przypisujesz jej wartość, musisz napisać nazwę tej zmiennej po znaku zachęty >>>, aby sprawdzić, jaką wartość przechowuje.

Przyjrzyj się działaniom przedstawionym poniżej. Będzie nawet lepiej, gdy krok po kroku samodzielnie wypróbujesz te przykłady w *swojej* powłoce. Pamiętaj tylko o tym, aby na końcu każdej instrukcji programu nacisnąć klawisz *Enter*, który informuje też powłokę, że instrukcję tę należy wykonać *od razu*.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
============= RESTART: C:\Users\lsuma\Desktop\Python\R01\odd.py =============
Ta minuta wydaje się dość nieparzysta.
>>>
============= RESTART: C:\Users\lsuma\Desktop\Python\R01\odd.py =============
Minuta parzysta.
>>>
>>> print('Cześć, mamo!')
Cześć, mamo!
>>>
>>> 21+21
42
>>>
>>> ultimate_answer = 21+21
>>> ultimate_answer
42
>>> |
                                                                  Ln: 19  Col: 4
```

W wyniku wykonania tej instrukcji kodu powłoka wyświetla na ekranie komunikat (tylko nie zapomnij nacisnąć klawisza Enter).

Gdy przeprowadzasz jakąś operację obliczeniową, powłoka wyświetla wartość stanowiącą jej wynik (po tym, jak naciśniesz klawisz Enter).

Operacja przypisania wartości do zmiennej nie skutkuje wyświetleniem tej wartości. Aby ją zobaczyć na ekranie, musisz wyraźnie poprosić o to powłokę.

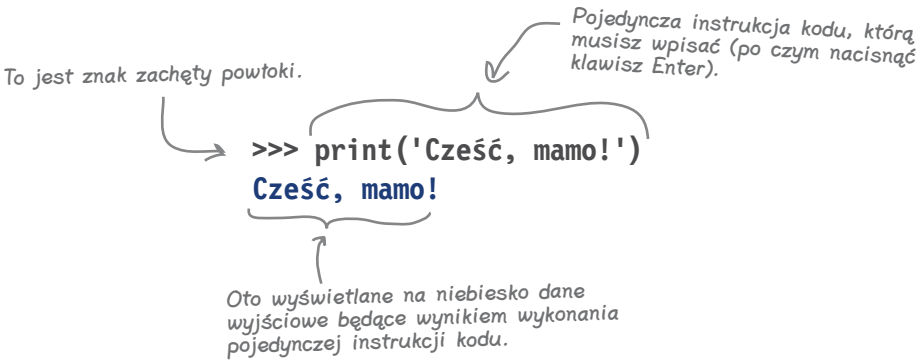
Eksperymentowanie w powłoce

Teraz, gdy wiesz już, że da się wpisać pojedynczą instrukcję Pythona po znaku zachęty >>> i wykonać ją bez zwłoki, możesz przystąpić do opracowywania kodu, który jest niezbędny, aby rozbudować Twój program.

Oto co powinien robić Twój nowy kod:

- ☐ **Powtarzać** działanie określoną liczbę razy. Zdecydowaliśmy już, że skorzystamy tu z zapewnianej przez język Python pętli `for`.
- ☐ **Wstrzymywać** program na wskazaną liczbę sekund. Umożliwi nam to funkcja `sleep` należąca do modułu `time` biblioteki standardowej.
- ☐ **Generować** liczbę losową należącą do przedziału wyznaczonego przez dwie zadane wartości. Sztuczkę tę pomoże nam wykonać funkcja `randint`, należąca do modułu `random` biblioteki standardowej.

Zamiast pokazywać Ci całe zrzuty okna programu IDLE, będziemy jedynie prezentować zawartość wiersza poleceń powłoki (wraz ze znakiem zachęty >>>) oraz dane wyjściowe, które będą w nim wyświetlane. A konkretnie: zamiast pojawiających się wcześniej w tym rozdziale zrzutów ekranu od tej chwili będziesz oglądał coś takiego jak poniżej.



Na kilku następnych stronach będziemy się zajmować eksperymentami, które mają przybliżyć nas do odpowiedzi na pytanie, jak dodać do naszego programu trzy funkcje wymienione powyżej. Będziemy się *bawić* kodem w wierszu poleceń, aż uda nam się dokładnie określić postać instrukcji, których w tym celu potrzebujemy. Zostaw na razie w spokoju kod zapisany w pliku *odd.py*, a następnie uaktywnij okno powłoki, przechodząc do niego. Kursor tekstowy powinien mrugać na prawo od znaku zachęty >>>, czekając cierpliwie, aż wpiszesz tam jakiś kod.

Przejdź do następnej strony, gdy będziesz gotów. Niech zaczną się eksperymenty.

Iterowanie po sekwencji obiektów

Poinformowaliśmy Cię wcześniej, że zamierzamy tu skorzystać z oferowanej przez język Python pętli `for`. Pętla `for` jest *doskonała* do sterowania powtarzaniem czynności, jeśli wiesz z góry, ilu iteracji potrzebujesz. (Jeśli tego nie wiesz, radzimy Ci zastosować pętlę `while`, ale omówienie szczegółów dotyczących alternatywnych konstrukcji umożliwiających tworzenie pętli odłożymy do czasu, gdy rzeczywiście będą nam one potrzebne). Na tym etapie potrzebujemy jedynie pętli `for`, zobaczmy ją więc w akcji, korzystając z wiersza poleceń powłoki.

Zaprezentujemy tu trzy typowe sposoby zastosowania pętli `for`. Sprawdźmy, który z nich najlepiej spełni nasze oczekiwania.

Używaj pętli for, gdy znana jest liczba powtórzeń czynności.

Przykład użycia nr 1. Przedstawiona poniżej pętla `for` przyjmuje listę liczb i powtarza operację wyświetlenia na ekranie bieżącej wartości po jednym razie dla każdej liczby znajdującej się na liście. Robiąc to, przypisuje każdą liczbę po kolei do *zmiennej iteracyjnej pętli*, która w naszym kodzie otrzymała nazwę `i`.

Z uwagi na to, że kod ten składa się z więcej niż jednego wiersza, powłoka automatycznie dodaje wcięcie, gdy tylko naciśniesz klawisz *Enter* po wpisaniu dwukropka. Aby powiadomić powłokę, że zakończyłeś wprowadzanie kodu, *dwukrotnie* naciśnij klawisz *Enter* na końcu zestawu pętli.

```
>>> for i in [1, 2, 3]:
        print(i)

1
2
3
```

W tym przykładzie użyliśmy „i" w roli zmiennej iteracyjnej pętli, ale równie dobrze mogłaby ona mieć właściwie dowolną inną nazwę. Nie zmienia to jednak faktu, że „i", „j" oraz „k" są niesamowicie popularnymi nazwami wykorzystywanymi przez programistów w tego typu sytuacjach.

Jako że mamy tu do czynienia z zestawem, po wprowadzeniu tego kodu musisz nacisnąć klawisz Enter DWUKROTNIE, aby zakończyć instrukcję i zobaczyć wynik jej wykonania.

Zwróć uwagę na *wcięcie* i *dwukropek*. Podobnie jak ma to miejsce w przypadku instrukcji `if`, kod związany z instrukcją `for` musi być odpowiednio **wcięty**.

Przykład użycia nr 2. Przedstawiona poniżej pętla `for` iteruje po łańcuchu znakowym, przetwarzając po jednym znaku należącym do tego łańcucha w czasie każdej iteracji. Rozwiązanie to działa, ponieważ łańcuch znakowy jest w języku Python **sekwencją** (ciągiem). Sekwencja to uporządkowana kolekcja obiektów (w książce tej naoglądasz się jeszcze mnóstwa przykładów rozmaitych sekwencji), a w Pythonie można iterować po każdej sekwencji, korzystając z interpretera.

```
>>> for ch in "Hej!":
        print(ch)

H
e
j
!
```

Python jest wystarczająco sprytny, aby stwierdzić, że ten łańcuch znakowy powinien być iterowany po jednym znaku naraz (dlatego też użyliśmy tu „ch" — jak „character", czyli znaku — jako nazwy zmiennej pętli).

Sekwencja to uporządkowana kolekcja obiektów.

Nigdzie nie musiałeś poinformować pętli `for`, *jak długi jest łańcuch znakowy*. Python jest wystarczająco sprytny, aby stwierdzić, kiedy *kończy się* łańcuch, i odpowiednio przerwać (czyli zakończyć) działanie pętli `for` w Twoim imieniu, gdy tylko wyczerpią się wszystkie obiekty należące do sekwencji.

Iterowanie określoną liczbę razy

Oprócz tego, że pętli `for` można używać do iterowania po sekwencji, umożliwia ona też bardziej precyzyjne określanie liczby powtórzeń dzięki wbudowanej funkcji o nazwie `range`.

Przyjrzymy się kolejnemu przykładowi zastosowania pętli `for`, w połączeniu z funkcją `range`.

Przykład użycia nr 3. W swojej najprostszej postaci funkcja `range` przyjmuje pojedynczy argument będący liczbą całkowitą, która określa, ile razy ma się wykonać pętla `for` (inne przykłady użycia tej funkcji przedstawimy w dalszej części niniejszej książki). W przypadku pętli pokazanej poniżej korzystamy z funkcji `range`, aby wygenerować listę liczb, które są pojedynczo przypisywane do zmiennej `num`.

```
>>> for num in range(5):
            print('Python rządzi!')
Python rządzi!
Python rządzi!
Python rządzi!
Python rządzi!
Python rządzi!
```

Poprosiliśmy o zbiór pięciu liczb, dlatego nastąpiło pięć powtórzeń, dzięki czemu wyświetlonych zostało pięć komunikatów. Pamiętaj, że aby uruchomić kod zawierający zestaw, trzeba nacisnąć klawisz Enter dwa razy.

Pętla `for` *nie skorzystała* ze zmiennej iteracyjnej `num` *w żadnym miejscu* w swoim zestawie. Nie spowodowało to zgłoszenia błędu, co jest całkiem normalne, ponieważ to Ty (jako programista) decydujesz, czy zmienna `num` ma być w jakiś sposób przetwarzana w zestawie, czy też nie. W tym przypadku brak działań na `num` nie stanowi dla nas najmniejszego problemu.

Istotnie, mamy. Zadanie nr 1 zakończone.

Przedstawione trzy przykłady zastosowania pętli `for` języka Python pokazują, że jest ona dokładnie tym, czego powinniśmy tu użyć, skorzystajmy więc z metody zaprezentowanej w **Przykładzie użycia nr 3**, aby powtórzyć działanie pętli `for` *określoną liczbę razy*.

Zastosowanie rezultatu Zadania nr 1 w naszym kodzie

Poniżej widać, jak wyglądał nasz kod w oknie edycji środowiska IDLE *przed* rozpoczęciem prac nad Zadaniem nr 1.

odd.py - C:\Users\lsuma\Desktop\Python\R01\odd.py (3.6.0)

File Edit Format Run Options Window Help

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

Ln: 14 Col: 0

To kod, którego wykonanie chcemy powtarzać.

Wiesz, że możesz skorzystać z pętli `for`, aby pięciokrotnie powtórzyć wykonanie tych pięciu wierszy kodu znajdujących się na końcu programu. W tym celu powinny one zostać odpowiednio **wcięte** pod instrukcją `for`, ponieważ utworzą zestaw tej pętli. A dokładniej: do każdego wiersza tego fragmentu kodu trzeba zastosować *jedno* wcięcie. Nie czuj się jednak zobowiązany do przeprowadzania tej operacji na pojedynczych wierszach, lecz pozwól, aby środowisko IDLE zrobiło to dla całego zestawu *za jednym zamachem*.

Zacznij od zaznaczenia odpowiednich wierszy kodu za pomocą myszy, tak jak zostało to pokazane poniżej.

odd.py - C:\Users\lsuma\Desktop\Python\R01\odd.py (3.6.0)

File Edit Format Run Options Window Help

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

Ln: 14 Col: 0

Użyj myszy w celu zaznaczenia wierszy kodu, które chcesz wciąć.

Wcinaj zestawy, korzystając z funkcji IDLE

Po zaznaczeniu sześciu właściwych wierszy kodu wybierz z menu *Format* okna edycji IDLE pozycję *Indent Region* (wetnij region). Cały zestaw zostanie wówczas przesunięty w prawo o jeden poziom wcięcia.

odd.py - C:\Users\lsuma\Desktop\Python\R01\odd.py (3.6.0)

File Edit Format Run Options Window Help

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

   right_this_minute = datetime.today().minute

   if right_this_minute in odds:
      print("Ta minuta wydaje się dość nieparzysta.")
   else:
      print("Minuta parzysta.")
```

Ln: 14 Col: 0

Należące do menu Format polecenie Indent Region umożliwia zastosowanie wcięcia dla wszystkich zaznaczonych wierszy kodu za jednym razem.

Zwróć uwagę, że IDLE oferuje również w swoim menu polecenie *Dedent Region* (usuń wcięcie regionu), które umożliwia usuwanie wcięcia zestawów. Zauważ też, że obydwa te polecenia mają swoje skróty klawiszowe, które różnią się nieco w zależności od systemu operacyjnego, z jakiego korzystasz. Poświęć *teraz* chwilę, aby zapamiętać skróty właściwe dla Twojego systemu (ponieważ będziesz z nich korzystał niemal bez przerwy). Jeśli zastosowałeś już odpowiednie wcięcie, czas dodać samą pętlę `for`.

odd.py - C:\Users\lsuma\Desktop\Python\R01\odd.py (3.6.0)

File Edit Format Run Options Window Help

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
   right_this_minute = datetime.today().minute

   if right_this_minute in odds:
      print("Ta minuta wydaje się dość nieparzysta.")
   else:
      print("Minuta parzysta.")
```

Ln: 15 Col: 0

Dodaj wiersz z pętlą for.

Zestaw pętli for jest prawidłowo wcięty.

Przygotowanie do wstrzymania wykonania

Przypomnijmy sobie, co nasz kod powinien robić:

- [x] **Powtarzać** działanie określoną liczbę razy.
- [ ] **Wstrzymywać** program na wskazaną liczbę sekund.
- [ ] **Generować** liczbę losową należącą do przedziału wyznaczonego przez dwie zadane wartości.

Jesteśmy już gotowi powrócić do powłoki i wypróbować nieco więcej kodu, który pomoże nam rozwiązać drugie zadanie: *wstrzymać program na wskazaną liczbę sekund*.

Zanim to jednak zrobimy, przypomnij sobie pierwszy wiersz naszego programu. W wierszu tym był importowany wyszczególniony submoduł należący do wskazanego modułu (choć w przypadku innego modułu równie dobrze mogłaby być to funkcja).

```
from datetime import datetime
```

Dzięki takiemu użyciu konstrukcji importowania masz w swoim programie dostęp do określonego submodułu. Możesz się do niego odwołać bez konieczności stosowania składni wykorzystującej notację kropkową.

To jeden ze sposobów importowania submodułu lub funkcji do programu. Inną, równie popularną metodą, jest importowanie modułu *bez* określania submodułu czy funkcji, której chce się użyć. Skorzystaj tu z tego drugiego sposobu, ponieważ będzie się on pojawiał w wielu programach Pythona, z którymi się zetkniesz w swojej karierze.

Jak wspomnieliśmy już wcześniej w tym rozdziale, funkcja `sleep` może wstrzymać wykonanie programu na określoną liczbę sekund i należy do modułu `time` biblioteki standardowej. **Zaimportujmy** ten moduł *najpierw*, nie wspominając jednak ani słowem o samej funkcji `sleep`.

```
>>> import time
>>>
```

Ta instrukcja informuje powłokę, że należy zaimportować moduł time.

Jeśli instrukcja `import` jest używana w taki sposób jak w pokazanym przykładzie z modułem `time`, otrzymujesz dostęp do wszystkich dobrodziejstw zapewnianych przez dany moduł bez konieczności wyraźnego *nazwania* tego, co miało zostać zaimportowane. Aby odwołać się do funkcji należącej do modułu zaimportowanego w ten sposób, użyj składni z notacją kropkową w celu wskazania nazwy tej funkcji.

Najpierw wskaż nazwę modułu (przed kropką).

To liczba sekund, które należy „przespać".

```
>>> time.sleep(5)
>>>
```

Wskaż funkcję, którą chcesz wywołać (po kropce).

Zauważ, że gdy wywołasz w ten sposób funkcję `sleep`, powłoka wstrzyma swoje działanie na pięć sekund i dopiero po tym czasie ponownie pojawi się znak zachęty >>>. Śmiało, *sprawdź to teraz*.

Zamieszanie z importowaniem

Python zapewnia dwa różne mechanizmy importowania? Czy nie jest to nieco mylące? Chwileczkę...

To świetne pytanie.

Gwoli ścisłości: w języku Python nie istnieją *dwa* mechanizmy importowania, ponieważ jest tylko *jedna* instrukcja `import`. Można jej jednak używać *na dwa sposoby*.

Pierwszy z nich, z którym mieliśmy do czynienia w przykładowym programie od samego początku, umożliwia zaimportowanie do **przestrzeni nazw** naszego programu nazwanego submodułu lub funkcji, co z kolei pozwala nam w każdej chwili wywoływać je bez konieczności *łączenia* z zaimportowanym modułem. (Pojęcie przestrzeni nazw jest w Pythonie bardzo ważne, ponieważ definiuje ona kontekst, w którym wykonuje się Twój kod. Na razie tyle na ten temat, szczegóły dotyczące przestrzeni nazw zostaną omówione w jednym z kolejnych rozdziałów książki).

W naszym przykładowym programie korzystamy z pierwszej metody importowania, a następnie odwołujemy się do submodułu `datetime`, posługując się składnią `datetime`, a *nie* konstrukcją `datetime.datetime`, jak miałoby to miejsce w przypadku drugiego sposobu.

Druga metoda użycia instrukcji `import` polega po prostu na zaimportowaniu całego modułu, tak jak zrobiliśmy to, eksperymentując z modułem `time`. Jeśli korzystamy z tego sposobu, musimy posłużyć się składnią z notacją kropkową, aby móc odwołać się do funkcji oferowanych przez moduł. Dlatego właśnie zastosowaliśmy konstrukcję `time.sleep()`.

Nie istnieją głupie pytania

P: Czy istnieje jeden prawidłowy sposób używania instrukcji `import`?

O: Zwykle sprowadza się to do osobistych preferencji programistów, jako że jedni wolą bardziej, a inni mniej szczegółowe podejście do tematu. Czasem zdarzają się jednak sytuacje, gdy w dwóch modułach (nazwijmy je A i B) występuje funkcja o tej samej nazwie (załóżmy, że będzie to F). Jeśli użyjesz w swoim kodzie instrukcji `from A import F` oraz `from B import F`, to skąd interpreter ma wiedzieć, którą wersję funkcji F powinien wywołać, gdy napiszesz `F()`? Jedyny sposób, aby tego uniknąć, polega na skorzystaniu w kodzie z niespecyficznych instrukcji `import` (a więc `import A` i `import B`), a następnie na wywołaniu konkretnej (tej, o którą Ci w danej chwili chodzi) wersji funkcji F, odpowiednio, za pomocą konstrukcji `A.F()` lub `B.F()`. Robiąc tak, unikniesz wszelkich wątpliwości i kłopotów.

Generowanie losowych liczb całkowitych w Pythonie

Choć kuszące wydaje się teraz dodanie instrukcji `import time` na początku naszego programu, a następnie wywołanie funkcji `time.sleep(5)` w zestawie pętli `for`, nie zrobimy tego jeszcze. Nie zakończyliśmy naszych eksperymentów. Wstrzymanie wykonania programu na pięć sekund nie wystarczy; musimy być w stanie wstrzymać działanie na *losowy odcinek czasu*. Mając to na uwadze, przypomnijmy sobie, co już osiągnęliśmy i co jeszcze zostało do zrobienia:

- ☑ **Powtarzać** działanie określoną liczbę razy.
- ☑ **Wstrzymywać** program na wskazaną liczbę sekund.
- ☐ **Generować** liczbę losową należącą do przedziału wyznaczonego przez dwie zadane wartości.

Gdy uda nam się już doprowadzić ostatnie zadanie do końca, będziemy mogli wrócić do kodu i z dużą pewnością siebie zmienić nasz program w taki sposób, aby uwzględniał wnioski, które wyciągnęliśmy z naszych eksperymentów. Nie dotarliśmy jeszcze jednak do tego miejsca, dlatego przyjrzyjmy się ostatniemu zadaniu, polegającemu na generowaniu liczby losowej.

Podobnie jak miało to miejsce w przypadku wstrzymywania działania programu, również tutaj z pomocą przyjdzie nam *biblioteka standardowa*, ponieważ zawiera moduł o nazwie `random`. Wiedząc tylko tyle, spróbujmy wykonać w powłoce kilka eksperymentów.

```
>>> import random
>>>
```

Użyj funkcji dir, aby zbadać obiekt.

Co teraz? Moglibyśmy spojrzeć w dokumentację Pythona lub poczytać jakieś encyklopedyczne wydawnictwo na temat tego języka... ale to odwróciłoby naszą uwagę od powłoki, nawet gdyby miało zająć tylko kilka chwil. Tak się szczęśliwie składa, że sama powłoka zapewnia pewne dodatkowe funkcje, które mogą nam tu pomóc. Funkcje te nie są przeznaczone do używania w kodzie programów, a jedynie do stosowania w wierszu poleceń. Pierwsza z nich to `dir`; jej zadaniem jest wyświetlanie wszystkich **atrybutów** związanych z dowolną rzeczą w Pythonie, w tym — z modułami.

```
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', ... 'randint',
'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle',
'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
```

Nazwa funkcji, której potrzebujemy, jest „zakopana" gdzieś pod koniec tej długiej listy.

To skrócona lista. Ta, którą zobaczysz w oknie powłoki, będzie znacznie dłuższa.

Ta lista zawiera mnóstwo pozycji. Nas interesuje funkcja `randint`. Aby dowiedzieć się o niej więcej, poprośmy powłokę o **pomoc**.

Proszenie interpretera o pomoc

Gdy znasz już nazwę jakiejś rzeczy, możesz poprosić powłokę o **pomoc**. Gdy to zrobisz, w oknie pojawi się fragment dokumentacji Pythona związany z interesującą Cię nazwą.

Zobaczmy ten mechanizm w akcji, prosząc w wierszu poleceń o **pomoc** dotyczącą funkcji `randint` należącej do modułu `random`.

Użyj funkcji help, aby zapoznać się z dokumentacją Pythona.

Poproś o pomoc w wierszu poleceń...

```
>>> help(random.randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

...i zobacz odpowiednią dokumentację bezpośrednio w oknie powłoki.

Pobieżna lektura wyświetlonego fragmentu dokumentacji funkcji `randint` potwierdza to, co powinniśmy wiedzieć: gdy podamy jej dwie liczby całkowite, zwróci nam losową liczbę z ograniczonego przez nie przedziału (włącznie z nimi samymi).

Kilka ostatnich, przedstawionych poniżej eksperymentów z funkcją `randint` w wierszu poleceń pokazuje nam jej działanie.

Dla maniaków

Ostatnie instrukcje wprowadzone w wierszu poleceń środowiska IDLE (po znaku zachęty >>>) możesz przywołać z powrotem, korzystając ze skrótu klawiszowego *Alt+P* (w przypadku systemów *Linux* i *Windows*) lub *Ctrl+P* (w przypadku systemu *Mac OS X*). Łatwo to zapamiętać: wyobraź sobie po prostu, że klawisz litery *P* reprezentuje skrót od słowa „poprzedni".

```
>>> random.randint(1, 60)
27
>>> random.randint(1, 60)
34
>>> random.randint(1, 60)
46
```

Gdy wykonasz przedstawione tu instrukcje, na ekranie możesz zobaczyć zupełnie inne liczby; mogą się one różnić od naszych, ponieważ wartości całkowite zwracane przez funkcję randint są generowane losowo.

Jako że zaimportowałeś moduł random, korzystając z instrukcji import random, musisz pamiętać, aby wywołanie funkcji randint poprzedzić nazwą modułu i kropką. Z tego powodu odpowiednia instrukcja powinna mieć postać random.randint(), a nie randint().

Dysponując tą wiedzą, możesz już z satysfakcją przystąpić do odhaczenia ostatniego zadania na naszej liście. Wiesz już, jak generować liczby losowe należące do przedziału wyznaczanego przez dwie podane wartości.

- ✔ **Generować** liczbę losową należącą do przedziału wyznaczonego przez dwie zadane wartości.

Już czas powrócić do naszego programu i wprowadzić planowane zmiany.

Przeglądanie wyników naszych eksperymentów

Zanim przejdziesz dalej i zaczniesz zmieniać swój program, przeprowadźmy szybki przegląd efektu naszych eksperymentów w powłoce.

Zaczęliśmy od utworzenia pętli `for`, która została wykonana pięć razy.

```
>>> for num in range(5):
        print('Python rządzi!')
Python rządzi!
Python rządzi!
Python rządzi!
Python rządzi!
Python rządzi!
```

Poprosiliśmy o zbiór pięciu liczb, dlatego nastąpiło pięć powtórzeń, dzięki czemu wyświetlonych zostało pięć komunikatów.

Następnie użyliśmy należącej do modułu `time` funkcji `sleep` w celu wstrzymania wykonania naszego kodu na określoną liczbę sekund.

```
>>> import time
>>> time.sleep(5)
```

Powłoka importuje moduł time, dzięki czemu możemy wywołać funkcję sleep.

Na koniec eksperymentowaliśmy z funkcją `randint` (wchodzącą w skład modułu `random`) w celu generowania losowych liczb całkowitych należących do zadanego przedziału.

```
>>> import random
>>> random.randint(1, 60)
12
>>> random.randint(1, 60)
42
>>> random.randint(1, 60)
17
```

Zauważ, że ponownie wygenerowane zostały różne liczby całkowite, ponieważ funkcja randint zwraca inną wartość losową za każdym razem, gdy zostaje wywołana.

Możemy teraz złożyć wszystkie te klocki w całość i zmodyfikować nasz program.

Przypomnijmy sobie, co zdecydowaliśmy się zrobić we wcześniejszej części tego rozdziału: sprawić, żeby program iterował, wykonując „kod sprawdzania minuty” oraz instrukcję `if/else` pięciokrotnie, a także wstrzymywać działanie na losową liczbę sekund pomiędzy poszczególnymi iteracjami. W wyniku tego zanim program ostatecznie zakończy działanie, na ekranie powinno się pojawić pięć komunikatów.

Magnesiki do eksperymentów z kodem

Na podstawie specyfikacji znajdującej się na poprzednich stronach, jak również na bazie wyników naszych eksperymentów ruszyliśmy do przodu i wykonaliśmy za Ciebie trochę niezbędnej roboty. Jednak gdy układaliśmy nasze magnesiki z kodem na lodówce (nie pytaj o szczegóły), ktoś trzasnął drzwiami, skutkiem czego niektóre fragmenty kodu rozsypały się po podłodze.

Twoim zadaniem jest ponowne poskładanie wszystkiego w sensowną całość, tak abyśmy mogli uruchomić nową wersję naszego programu i upewnić się, że działa on zgodnie z założeniami.

Zdecyduj, w których miejscach wyznaczanych przez kropkowane linie mają się znaleźć poszczególne magnesiki z kodem.

```
from datetime import datetime
.........................
.........................

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
         21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
         41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

.........................
   right_this_minute = datetime.today().minute

   if right_this_minute in odds:
      print("Ta minuta wydaje się dość nieparzysta.")
   else:
      print("Minuta parzysta.")
   wait_time = .........................
   .................(....................)
```

Gdzie powinny się znaleźć?

- `time.sleep`
- `import time`
- `wait_time`
- `import random`
- `for i in range(5):`
- `random.randint(1, 60)`

Magnesiki do eksperymentów z kodem. Rozwiązanie

Na podstawie przedstawionej wcześniej specyfikacji, jak również na bazie wyników naszych eksperymentów ruszyliśmy do przodu i wykonaliśmy za Ciebie trochę niezbędnej roboty. Jednak gdy układaliśmy nasze magnesiki z kodem na lodówce (nie pytaj o szczegóły), ktoś trzasnął drzwiami, skutkiem czego niektóre z fragmentów kodu rozsypały się po podłodze.

Twoim zadaniem było ponowne poskładanie wszystkiego w sensowną całość, tak abyśmy mogli uruchomić nową wersję naszego programu i upewnić się, że działa on zgodnie z założeniami.

Nie musisz umieszczać instrukcji importowania na początku swojego kodu, ale właśnie tak się robi zgodnie z konwencją, która jest ogólnie akceptowana przez programistów Pythona.

```
from datetime import datetime
import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("Ta minuta wydaje się dość nieparzysta.")
    else:
        print("Minuta parzysta.")
    wait_time =
    random.randint(1, 60)
    time.sleep
    (
    wait_time
    )
```

Pętla for wykonuje się DOKŁADNIE pięć razy.

Funkcja randint zwraca losową liczbę całkowitą; ta zostaje przypisana do nowej zmiennej o nazwie „wait_time", która...

...jest następnie wykorzystywana w wywołaniu funkcji sleep wstrzymującej wykonanie programu na losową liczbę sekund.

Cały ten kod jest wcięty pod instrukcją for, ponieważ stanowi jej zestaw. Pamiętaj, że w Pythonie nie używa się nawiasów klamrowych do ograniczania zestawów, lecz wyłącznie stosuje się odpowiednie wcięcia.

Jazda próbna

Spróbujmy uruchomić nasz zmodyfikowany program w środowisku IDLE, aby zobaczyć, co się stanie. Zmień odpowiednio swoją wersję pliku *odd.py*, a następnie zapisz kopię nowego programu w pliku o nazwie *odd2.py*. Po zrobieniu tego naciśnij klawisz *F5*, aby uruchomić swój kod.

Gdy naciśniesz klawisz F5, aby uruchomić ten kod...

odd2.py - C:/Users/lsuma/Desktop/Python/R01/odd2.py (3.6.0)

File Edit Format Run Options Window Help

```
from datetime import datetime

import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("Ta minuta wydaje się dość nieparzysta.")
    else:
        print("Minuta parzysta.")
    wait_time = random.randint(1, 60)
    time.sleep(wait_time)
```

Ln: 19 Col: 0

...powinieneś zobaczyć dane wyjściowe podobne do przedstawionych tutaj. Pamiętaj tylko, że na Twoim ekranie może to wyglądać nieco inaczej, ponieważ liczby losowe wygenerowane przez Twój program prawdopodobnie nie będą odpowiadały naszym wartościom.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
============= RESTART: C:/Users/lsuma/Desktop/Python/R01/odd2.py =============
Minuta parzysta.
Ta minuta wydaje się dość nieparzysta.
Minuta parzysta.
Ta minuta wydaje się dość nieparzysta.
Minuta parzysta.
>>>
```

Ln: 10 Col: 4

Nie martw się, gdy zobaczysz inną listę komunikatów niż te, które widać tutaj. Powinieneś ujrzeć pięć komunikatów, ponieważ to właśnie tyle razy wykona się kod znajdujący się w pętli.

Odświeżanie tego, co już wiemy

Doprowadziwszy zmiany do końca, zatrzymajmy się raz jeszcze na chwilę, aby przejrzeć nowe rzeczy, których dowiedzieliśmy się o języku Python z ostatnich 15 stron książki.

CELNE SPOSTRZEŻENIA

- Starając się wymyślić kod, który jest potrzebny do rozwiązania określonego problemu, programiści Pythona często decydują się na przeprowadzanie eksperymentów na fragmentach kodu w powłoce.
- Jeśli widzisz znak zachęty `>>>`, korzystasz z okna powłoki Pythona. Śmiało, wprowadź pojedynczą instrukcję języka i zobacz, co się stanie, gdy zostanie ona wykonana.
- Powłoka przechwytuje Twój wiersz kodu i przesyła go do interpretera, który go wykonuje. Wszystkie wyniki są zwracane do powłoki, a następnie wyświetlane na ekranie.
- Pętla `for` może być używana do powtarzania działań określoną liczbę razy. Jeśli z góry wiesz, ile razy należy iterować, skorzystaj właśnie z pętli `for`.
- Jeśli nie wiesz z góry, ile razy trzeba iterować, skorzystaj z zapewnianej przez Pythona pętli `while` (z którą jeszcze nie mieliśmy tu do czynienia, ale — nie martw się — wkrótce ujrzymy ją w akcji).
- Pętla `for` umożliwia iterowanie po sekwencji (takiej jak lista czy łańcuch znakowy), jak również wykonywanie pewnych operacji określoną liczbę razy (dzięki zastosowaniu funkcji `range`).
- Jeśli musisz wstrzymać wykonywanie programu na określoną liczbę sekund, skorzystaj z funkcji `sleep` zapewnianej przez moduł `time` należący do biblioteki standardowej.
- Możesz zaimportować wybraną funkcję należącą do modułu. Na przykład instrukcja `from time import sleep` powoduje zaimportowanie funkcji `sleep` z modułu `time`, umożliwiając Ci wywołanie jej bez konieczności odwoływania się do nazwy modułu, do którego należy.
- Jeśli po prostu zaimportujesz moduł, na przykład za pomocą instrukcji `import time`, wywołania należących do niego funkcji będziesz musiał poprzedzać jego nazwą, na przykład tak: `time.sleep()`.
- Moduł `random` zawiera bardzo przydatną funkcję o nazwie `randint`, która umożliwia generowanie losowych liczb całkowitych należących do określonego przedziału.
- Powłoka oferuje dwie interaktywne funkcje, które można wywoływać z poziomu wiersza poleceń. Funkcja `dir` wyświetla listę atrybutów obiektu, a funkcja `help` zapewnia dostęp do dokumentacji Pythona.

Nie istnieją głupie pytania

P: Czy muszę pamiętać to wszystko?

O: Nie, i nie świruj, jeśli Twój mózg wzbrania się przed przyjęciem wszystkiego, co widziałeś do tej pory. To dopiero pierwszy rozdział i zaplanowaliśmy go tak, aby stanowił szybkie wprowadzenie do świata programowania w języku Python. Jeśli nadążasz za tym, co się dzieje w przedstawionym tu kodzie, to świetnie sobie radzisz.

Kilka wierszy kodu robi bardzo wiele

Zgadza się, ale jesteśmy teraz na fali.

Prawdą jest, że do tej pory udało nam się zaledwie delikatnie liznąć język Python. Jednak to, co zobaczyliśmy, było bardzo przydatne.

To, z czym mieliśmy już do czynienia, pomogło nam pokazać jedną z najlepszych cech języka Python, a mianowicie to, że *kilka wierszy kodu robi bardzo wiele*. Innym powodem sławy, jaką cieszy się ten język, jest to, że *kod Pythona łatwo się czyta*.

Podejmując próbę udowodnienia Ci, jak bardzo jest to łatwe, na następnej stronie przedstawimy całkiem inny program, który będziesz w stanie zrozumieć dzięki wiedzy o Pythonie, którą udało Ci się zgromadzić do tej pory.

Kto ma ochotę na dobre, schłodzone piwo?

Pisanie poważnej aplikacji biznesowej

Chyląc czoła przed książką *Java. Rusz głową!*, przyjrzyjmy się pythonowej wersji klasycznej, pierwszej poważnej aplikacji biznesowej, czyli piosence o piwie.

Poniżej przedstawione zostało okno zawierające pythonową wersję kodu piosenki o piwie. Poza nieco odmiennym od dotychczasowego sposobem użycia funkcji `range` (który pokrótce omówimy) większość pokazanego tu kodu powinna być dla Ciebie dość czytelna. Kod ten znajduje się w oknie edycji środowiska IDLE, a widoczne poniżej niego okno powłoki zawiera ostatnie dane wyjściowe wyświetlone przez program.

beersong.py - C:/Users/lsuma/Desktop/Python/R01/beersong.py (3...

File Edit Format Run Options Window Help

```
word = "butelki"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "piwa na ścianie.")
    print(beer_num, word, "piwa.")
    print("Jedną weź.")
    print("Podaj w koło.")
    if beer_num == 1:
        print("Nie ma już butelek piwa na ścianie.")
    else:
        new_num = beer_num - 1
        if new_num == 1:
            word = "butelka"
        print(new_num, word, "piwa na
    print()
```

Uruchomienie tego kodu skutkuje pojawieniem się takich danych wyjściowych w powłoce.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
3 butelki piwa na ścianie.
3 butelki piwa.
Jedną weź.
Podaj w koło.
2 butelki piwa na ścianie.

2 butelki piwa na ścianie.
2 butelki piwa.
Jedną weź.
Podaj w koło.
1 butelka piwa na ścianie.

1 butelka piwa na ścianie.
1 butelka piwa.
Jedną weź.
Podaj w koło.
Nie ma już butelek piwa na ścianie.

>>>
```

Ln: 599 Col: 4

Radzenie sobie z całym tym piwem...

Gdy przedstawiony przed chwilą kod wprowadzisz w oknie edycji środowiska IDLE, zapiszesz w pliku i naciśniesz klawisz *F5*, w oknie powłoki pojawi się mnóstwo danych wyjściowych. Obok pokazaliśmy jedynie niewielką część tych danych, jako że tekst piosenki o piwie zaczyna się od 99 butelek, a potem następuje mozolne odliczanie aż do momentu, gdy w ogóle nie ma już piwa. Jedynym prawdziwym trikiem w tym kodzie jest sposób poradzenia sobie z tym „odliczaniem”, więc przyjrzyjmy się, jak on działa, zanim przystąpimy do szczegółowego rozpracowywania całego kodu programu.

Kod Pythona łatwo się czyta

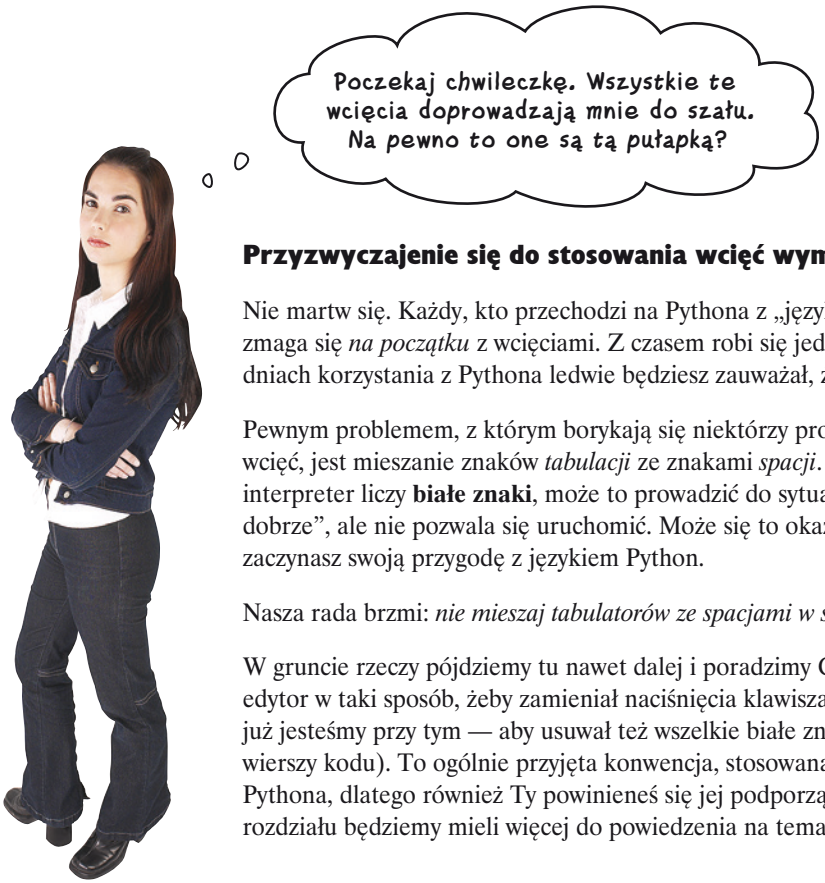
Nie ma żadnego haczyka!

Gdy większość programistów rozpoczynających naukę Pythona widzi po raz pierwszy kod podobny do tego, który wyświetla tekst piosenki o piwie, zakłada, że coś musi być tu nie tak.

Na pewno gdzieś zastawiono tu jakąś pułapkę, zgadza się?

Nie, nie ma żadnej pułapki. To nie przypadek, że kod Pythona łatwo się czyta: język ten został specjalnie zaprojektowany w taki sposób, aby osiągnąć ten cel. *Guido van Rossum*, twórca Pythona, chciał opracować potężne narzędzie programistyczne, za pomocą którego można tworzyć kod łatwy w utrzymaniu, a to oznacza, że kod ten musi się również dać łatwo czytać.

Czy wcięcia doprowadzają Cię do szału?



Przyzwyczajenie się do stosowania wcięć wymaga trochę czasu.

Nie martw się. Każdy, kto przechodzi na Pythona z „języka klamrowo-nawiasowego", zmaga się *na początku* z wcięciami. Z czasem robi się jednak łatwiej. Po dniu lub dwóch dniach korzystania z Pythona ledwie będziesz zauważał, że wcinasz swoje zestawy.

Pewnym problemem, z którym borykają się niektórzy programiści przy stosowaniu wcięć, jest mieszanie znaków *tabulacji* ze znakami *spacji*. Z uwagi na sposób, w jaki interpreter liczy **białe znaki**, może to prowadzić do sytuacji, w której kod „wygląda dobrze", ale nie pozwala się uruchomić. Może się to okazać bardzo frustrujące, gdy zaczynasz swoją przygodę z językiem Python.

Nasza rada brzmi: *nie mieszaj tabulatorów ze spacjami w swoim kodzie w języku Python*.

W gruncie rzeczy pójdziemy tu nawet dalej i poradzimy Ci, abyś skonfigurował swój edytor w taki sposób, żeby zamieniał naciśnięcia klawisza *Tab* na *cztery spacje*[1] (a gdy już jesteśmy przy tym — aby usuwał też wszelkie białe znaki występujące na końcu wierszy kodu). To ogólnie przyjęta konwencja, stosowana przez wielu programistów Pythona, dlatego również Ty powinieneś się jej podporządkować. Na końcu tego rozdziału będziemy mieli więcej do powiedzenia na temat radzenia sobie z wcięciami.

Powrót do kodu piosenki o piwie

Gdy spojrzysz na wywołanie funkcji `range` w kodzie wyświetlającym tekst piosenki o piwie, zauważysz, że przyjmuje on *trzy* argumenty, a nie tylko jeden (jak miało to miejsce w przypadku naszego pierwszego przykładowego programu).

Przyjrzyj się bliżej i bez zaglądania do wyjaśnień przedstawionych na następnej stronie sprawdź, czy jesteś w stanie wydedukować, w jaki sposób działa to wywołanie funkcji `range`.

```
word = "butelki"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "piwa na ś
    print(beer_num, word, "piwa.")
    print("Jedną wź
```

To coś nowego: wywołanie funkcji range przyjmuje trzy argumenty zamiast jednego.

[1] W tej książce, podobnie jak w innych publikacjach wydawnictwa Helion, a także w dołączonych do niej plikach kodów, w celu zapewnienia większej zwartości kodu stosowana jest konwencja wcięć o szerokości trzech spacji — *przyp. tłum.*

Proszenie interpretera o pomoc dotyczącą funkcji

Pamiętaj, że gdy masz jakikolwiek problem z Pythonem, zawsze możesz poprosić powłokę o **pomoc**. Zwróćmy się więc do niej z prośbą o informacje na temat funkcji `range`.

Gdy zrobisz to w oknie IDLE, wyświetlona dokumentacja wypełni cały ekran i szybko przewinie go dalej. Będziesz zatem musiał przewinąć zawartość okna wstecz do miejsca, w którym poprosiłeś powłokę o pomoc (ponieważ to właśnie tam znajdują się najciekawsze informacje na temat funkcji `range`, co widać poniżej).

```
>>> help(range)
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return a sequence of numbers from start to stop by step.
            ...
```

Funkcja range może być wywołana na jeden z dwóch sposobów.

Wygląda na to, że dostaniemy to, czego nam potrzeba.

Rozpoczynanie, kończenie i kroczenie

Funkcja `range` to niejedyne miejsce, w którym spotkasz się z **początkiem**, **końcem** i **krokiem** (ang. *start*, *stop*, *step*), dlatego poświęcimy chwilę na opis znaczenia każdego z tych pojęć, zanim przyjrzymy się reprezentatywnym przykładom zastosowania funkcji (przedstawionym na następnej stronie).

1. **Wartość POCZĄTEK zapewnia kontrolę nad tym, GDZIE zakres się zaczyna.**
   Jak dotąd korzystaliśmy wyłącznie z jednoargumentowej wersji funkcji `range`, która — zgodnie z dokumentacją — spodziewa się otrzymać wartość określającą **koniec**. Jeśli nie poda się innych wartości, funkcja ta przyjmuje dla **początku** standardową wartość 0, możesz jednak wskazać jej inną, jeśli tylko chcesz. Gdy to zrobisz, *musisz* określić wartość **końca**. W ten sposób wywołanie staje się wieloargumentowe.

2. **Wartość KONIEC zapewnia kontrolę nad tym, KIEDY zakres się kończy.**
   Mieliśmy już z nią do czynienia, gdy w naszym kodzie korzystaliśmy z wywołania o postaci `range(5)`. Zwróć uwagę na to, że wygenerowany zakres *nigdy* nie zawiera samej wartości **końcowej**, mamy tu więc przypadek zbioru zawierającego-wszystko-do-lecz-nie-włącznie-z **końcem**.

3. **Wartość KROK zapewnia kontrolę nad tym, JAK zakres jest generowany.**
   Określając wartości **początku** i **końca**, możesz również (opcjonalnie) wskazać wartość definiującą **krok**. Standardową wartością **kroku** jest 1; mówi ona funkcji `range`, że wygenerowana ma być każda kolejna wartość **różniąca się** od poprzedniej o 1, a więc: 0, 1, 2, 3, 4 i tak dalej. Wartość **kroku** możesz określić dowolnie, tak aby odpowiednio zdefiniować tę różnicę. Możesz również sprawić, aby **krok** miał wartość ujemną, dostosowując w ten sposób do swoich potrzeb *kierunek* generowanego zakresu.

Eksperymentowanie z zakresami

Teraz, gdy wiesz już co nieco na temat **początku**, **końca** i **kroku**, poeksperymentujmy trochę w powłoce, aby dowiedzieć się, jak możemy skorzystać z funkcji `range` w celu wygenerowania wielu różnych zakresów liczb całkowitych.

W trzymaniu ręki na pulsie pomoże nam inna funkcja, `list`, umożliwiająca przetwarzanie danych wyjściowych funkcji `range` w czytelne dla nas listy, które będziemy mogli zobaczyć na ekranie.

```
>>> range(5)
range(0, 5)
```

W ten sposób korzystaliśmy z funkcji range w naszym pierwszym programie.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Przekierowanie danych wyjściowych z funkcji range do funkcji list skutkuje wyświetleniem ich w postaci listy.

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

Możemy dostosować wartości POCZĄTKU i KOŃCA używane przez funkcję range.

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Możliwe jest też dostosowanie wartości KROK.

```
>>> list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

Robi się naprawdę ciekawie, gdy dostosujesz kierunek zakresu, korzystając z ujemnej wartości KROKU.

```
>>> list(range(10, 0, 2))
[]
```

Python nie zabroni Ci robić głupot. Jeśli użyta przez Ciebie wartość POCZĄTKU jest WIĘKSZA niż wartość KOŃCA, a wartość KROKU jest dodatnia, nie otrzymasz nic (w tym przypadku dostaniesz pustą listę).

```
>>> list(range(99, 0, -1))
[99, 98, 97, 96, 95, 94, 93, 92,  ...  5, 4, 3, 2, 1]
```

Po wykonaniu wszystkich naszych eksperymentów dotarliśmy do takiego wywołania funkcji `range` (pokazanego powyżej jako ostatnie), dzięki któremu udało nam się uzyskać listę wartości od 99 do 1 (a więc dokładnie tego, które jest wykorzystywane w pętli `for` użytej w kodzie programu wyświetlającego piosenkę o piwie, jak można się przekonać poniżej).

```
word = "butelki"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "piwa na s
    print(beer_num, word, "piwa.")
    print("Jedną wź
```

To wywołanie funkcji range przyjmuje trzy argumenty: początek, koniec i krok.

Zaostrz ołówek

Poniżej ponownie zamieściliśmy kod przykładu z piwem. Wypełniliśmy nim całą stronę, abyś mógł spokojnie **skoncentrować** się na każdym wierszu tworzącym tę „poważną aplikację biznesową".

Weź ołówek i napisz w pustych miejscach, co — Twoim zdaniem — robią poszczególne wiersze kodu. Spróbuj wykonać to zadanie samodzielnie, *zanim* spojrzysz na nasze wnioski, które zamieściliśmy na następnej stronie. Żeby Cię zachęcić do pracy, skomentowaliśmy pierwszy wiersz kodu za Ciebie.

```
word = "butelki"
```

Przypisujemy wartość "butelki" (będącą łańcuchem znakowym) do nowej zmiennej o nazwie „word".

```
for beer_num in range(99, 0, -1):
```

.....

```
    print(beer_num, word, "piwa na ścianie.")
```

.....

```
    print(beer_num, word, "piwa.")


    print("Jedną weź.")

    print("Podaj w koło.")
```

.....

```
    if beer_num == 1:
```

.....

```
        print("Nie ma już butelek piwa na ścianie.")
```

.....

```
    else:
```

.....

```
        new_num = beer_num - 1
```

.....

```
        if new_num == 1:
```

.....

```
            word = "butelka"
```

.....

```
        print(new_num, word, "piwa na ścianie.")
```

.....

```
    print()
```

.....

Zaostrz ołówek
Rozwiązanie

Poniżej ponownie zamieściliśmy kod przykładu z piwem. Wypełniliśmy nim całą stronę, abyś mógł spokojnie **skoncentrować** się na każdym wierszu tworzącym tę „poważną aplikację biznesową".

Miałeś wziąć ołówek i napisać w pustych miejscach, co — Twoim zdaniem — robią poszczególne wiersze kodu. Żeby Cię zachęcić do pracy, skomentowaliśmy pierwszy wiersz kodu za Ciebie.

Jak Ci poszło? Czy Twoje wyjaśnienia przypominają nasze?

```
word = "butelki"
```

Przypisujemy wartość "butelki" (będącą łańcuchem znakowym) do nowej zmiennej o nazwie „word".

```
for beer_num in range(99, 0, -1):
```

Powtarzamy działanie określoną liczbę razy, od 99 w dół do zera. Używamy beer_num w roli zmiennej iteracyjnej pętli.

```
    print(beer_num, word, "piwa na ścianie.")

    print(beer_num, word, "piwa.")

    print("Jedną weź.")

    print("Podaj w koło.")
```

Te cztery wywołania funkcji print odpowiadają za wyświetlenie słów piosenki odpowiednich dla bieżącej iteracji pętli. W każdej iteracji wyświetlana jest jedna (cała) zwrotka. Zaczynamy tu od zwrotki z 99 butelkami, następnie zmniejszamy liczbę butelek, wyświetlamy kolejną zwrotkę i tak dalej.

```
    if beer_num == 1:
```

Tu następuje sprawdzenie, czy jesteśmy przy ostatniej krążącej w koło butelce..

```
        print("Nie ma już butelek piwa na ścianie.")
```

I jeśli jesteśmy, wyświetlamy zakończenie tekstu piosenki.

```
    else:
```

W przeciwnym razie..

```
        new_num = beer_num - 1
```

Zapamiętujemy numer następnego piwa w kolejnej zmiennej o nazwie „new_num".

```
        if new_num == 1:
```

Jeśli właśnie przymierzamy się do wypicia naszego ostatniego piwa...

```
            word = "butelka"
```

Zmieniamy wartość zmiennej word, aby ostatni wiersz tekstu piosenki miał sens.

```
        print(new_num, word, "piwa na ścianie.")
```

Uzupełniamy tekst zwrotki wyświetlanej w bieżącej iteracji.

```
    print()
```

Na końcu tej iteracji wyświetlamy pusty wiersz. Gdy wykonamy wszystkie iteracje, kończymy działanie naszego programu.

Nie zapomnij wypróbować kodu piosenki o piwie

Jeśli nie zrobiłeś tego do tej pory, wprowadź kod programu wyświetlającego tekst piosenki o piwie do okna edycji środowiska IDLE, zapisz go pod postacią pliku *beersong.py*, a następnie naciśnij klawisz *F5*. *Nie przechodź do następnego rozdziału, dopóki nie uda Ci się uruchomić tego programu*.

Nie istnieją głupie pytania

P: Ciągle widzę błędy, gdy próbuję uruchomić swój kod wyświetlający tekst piosenki o piwie. Ale program wygląda moim zdaniem dobrze, więc czuję się trochę sfrustrowany. Jakieś sugestie?

O: Pierwszą rzeczą, którą należy sprawdzić, jest poprawność wcięć. Jeśli Twoje wcięcia są w porządku, sprawdź, czy nie wymieszałeś w swoim kodzie znaków tabulacji ze spacjami. Pamiętaj, że kod będzie (Ci) się wydawał poprawnie sformatowany, ale interpreter odmówi jego uruchomienia. Jeśli podejrzewasz tego rodzaju problem, najszybciej będzie, gdy przeniesiesz kod do okna edycji IDLE, wybierzesz z menu programu polecenie *Edit/Select All* (edycja/zaznacz wszystko), a następnie polecenie *Format/Untabify Region* (format/usuń tabulacje w regionie). Jeśli zdarzyło Ci się wymieszać w swoim kodzie tabulacje ze spacjami, operacja ta za jednym zamachem zamieni wszystkie znaki tabulacji w znaki spacji (i wyeliminuje wszelkie problemy z wcięciami).

Możesz następnie zapisać swój kod w pliku i nacisnąć klawisz *F5*, aby ponownie spróbować go uruchomić. Jeśli nadal będzie to niemożliwe, sprawdź, czy Twój kod jest na pewno *dokładnie* taki sam jak ten, który zaprezentowaliśmy w tym rozdziale. Zwróć szczególną uwagę na wszelkie błędy pisowni, które mogłeś popełnić w nazwach swoich zmiennych.

P: Interpreter Pythona nie ostrzeże mnie, jeśli pomylę się i napiszę `nwe_num` zamiast `new_num`?

O: Nie, nie zrobi tego. Dopóki do zmiennej przypisana jest wartość, Python zakłada, że wiesz, co robisz, i kontynuuje wykonywanie Twojego kodu. To jednak coś, na co warto zwrócić uwagę, więc zachowaj czujność.

Podsumowanie tego, co już wiesz

Poniżej znajdziesz listę kilku nowych rzeczy, których dowiedziałeś się dzięki analizie (i uruchomieniu) przykładu z kodem wyświetlającym tekst piosenki o piwie.

CELNE SPOSTRZEŻENIA

- Do wcięć trzeba się przyzwyczaić i może to wymagać trochę czasu. Każdy programista rozpoczynający swoją przygodę z Pythonem w pewnym momencie narzeka na wcięcia, ale nie martw się, bo wkrótce przestaniesz nawet zauważać, że w ogóle je robisz.
- Jeśli jest jakaś rzecz, której powinieneś za wszelką cenę unikać, to jest to mieszanie tabulacji ze spacjami podczas wcinania swojego kodu Pythona. Oszczędź sobie problemów w przyszłości i po prostu tego nie rób.
- Funkcja `range` może przyjmować więcej niż jeden argument wywołania. Argumenty te umożliwiają określanie wartości początku i końca generowanego zakresu, jak również wartość kroku.
- Jako wartość kroku funkcji `range` można podać liczbę ujemną, co powoduje zmianę kierunku generowanego zakresu.

Co dalej, skoro skończyło się już całe piwo?

To już wszystko, jeśli chodzi o rozdział 1. W następnym dowiesz się co nieco o tym, w jaki sposób można manipulować danymi w Pythonie. W niniejszym rozdziale pobieżnie zapoznałeś się jedynie z **listami**, teraz przyszedł czas na nieco głębsze zanurzenie się w tym temacie.

Kod przykładów z rozdziału 1.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
   print("Ta minuta wydaje się dość nieparzysta.")
else:
   print("Minuta parzysta.")
```

Zaczęliśmy od programu odd.py, a następnie...

...rozbudowaliśmy jego kod, aby utworzyć program odd2.py, który wykonuje „kod sprawdzania minuty" pięciokrotnie (dzięki zastosowaniu instrukcji pętli for zapewnianej przez język Python).

```
from datetime import datetime

import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
   right_this_minute = datetime.today().minute
   if right_this_minute in odds:
      print("Ta minuta wydaje się dość nieparzysta.")
   else:
      print("Minuta parzysta.")
   wait_time = random.randint(1, 60)
   time.sleep(wait_time)
```

```
word = "butelki"
for beer_num in range(99, 0, -1):
   print(beer_num, word, "piwa na ścianie.")
   print(beer_num, word, "piwa.")
   print("Jedną weź.")
   print("Podaj w koło.")
   if beer_num == 1:
      print("Nie ma już butelek piwa na ścianie.")
   else:
      new_num = beer_num - 1
      if new_num == 1:
         word = "butelka"
      print(new_num, word, "piwa na ścianie.")
   print()
```

Zakończyliśmy ten rozdział pythonową wersją klasycznego przykładu wykorzystywanego w serii „Rusz głową!", czyli „piosenką o piwie". I tak, mamy świadomość, że trudno jest patrzeć na ten kod i nie śpiewać pod nosem... :)

Korzystanie z uporządkowanych danych

Wszystkie programy przetwarzają dane i programy Pythona nie są tu żadnym wyjątkiem.

Istotnie, rozejrzyj się tylko wokół: *dane są dosłownie wszędzie*. Bardzo znaczna część, jeśli nie wręcz większość programowania dotyczy danych: *pobierania* danych, *przetwarzania* danych, *rozumienia* danych. Aby efektywnie korzystać z danych, musisz mieć jakieś miejsce do ich *przechowywania* w czasie przetwarzania. Python błyszczy w tej kwestii na tle konkurencji dzięki (w znacznej mierze) temu, że oferuje dostęp do struktur danych o *szerokim zastosowaniu*: **list**, **słowników**, **krotek** oraz **zbiorów**. W niniejszym rozdziale przedstawimy Ci pokrótce wszystkie te cztery rodzaje struktur, a następnie poświęcimy większą część jego treści na zagłębienie się w tematykę **list** (pozostałe trzy zostaną dokładniej omówione w kolejnym rozdziale). Postanowiliśmy zaprezentować Ci struktury danych na stosunkowo wczesnym etapie, ponieważ większość rzeczy, które będziesz robił z Pythonem, będzie się kręcić wokół korzystania z danych.

Liczby, łańcuchy znakowe... i obiekty

Korzystanie z *pojedynczej* wartości wygląda w Pythonie dokładnie tak, jak mógłbyś się tego spodziewać. Sprawa sprowadza się tu do przypisania wartości do zmiennej. Wspomagając się powłoką, przyjrzyjmy się kilku przykładom, aby przypomnieć sobie, czego dowiedziałeś się w poprzednim rozdziale.

Liczby

Załóżmy, że zaimportowaliśmy już moduł `random`. Możemy teraz wywołać funkcję `random.randint`, aby wygenerować losową liczbę z przedziału od 1 do 60, która zostanie następnie przypisana do zmiennej `wait_time`. Z uwagi na to, że wygenerowana wartość jest **liczbą całkowitą**, właśnie taki typ przyjmie ta zmienna w naszym przypadku.

```
>>> wait_time = random.randint(1, 60)
>>> wait_time
26
```

Zauważ, że nigdzie nie musiałeś informować interpretera, że zmienna `wait_time` ma zawierać wartość całkowitą. *Przypisaliśmy* liczbę całkowitą do zmiennej, a interpreter sam zadbał już o wszystkie szczegóły (warto tu odnotować, że nie wszystkie języki programowania działają w ten sposób).

Zmienna przyjmuje typ wartości, która zostaje do niej przypisana.

Łańcuchy znakowe

Gdy do zmiennej przypisujesz łańcuch znakowy, dzieje się dokładnie to samo, co powyżej: to interpreter troszczy się o wszelkie szczegóły. Również w takim przypadku nie musimy z góry deklarować, że zmienna `word` używana w tym przykładzie będzie zawierała **łańcuch znakowy**.

```
>>> word = "butelki"
>>> word
'butelki'
```

Ta możliwość *dynamicznego* przypisywania wartości do zmiennej ma zasadnicze znaczenie dla pythonowego systemu zmiennych i typów. W gruncie rzeczy sprawy mają jeszcze bardziej ogólny charakter, ponieważ do zmiennej można w Pythonie przypisać dosłownie *cokolwiek*.

Obiekty

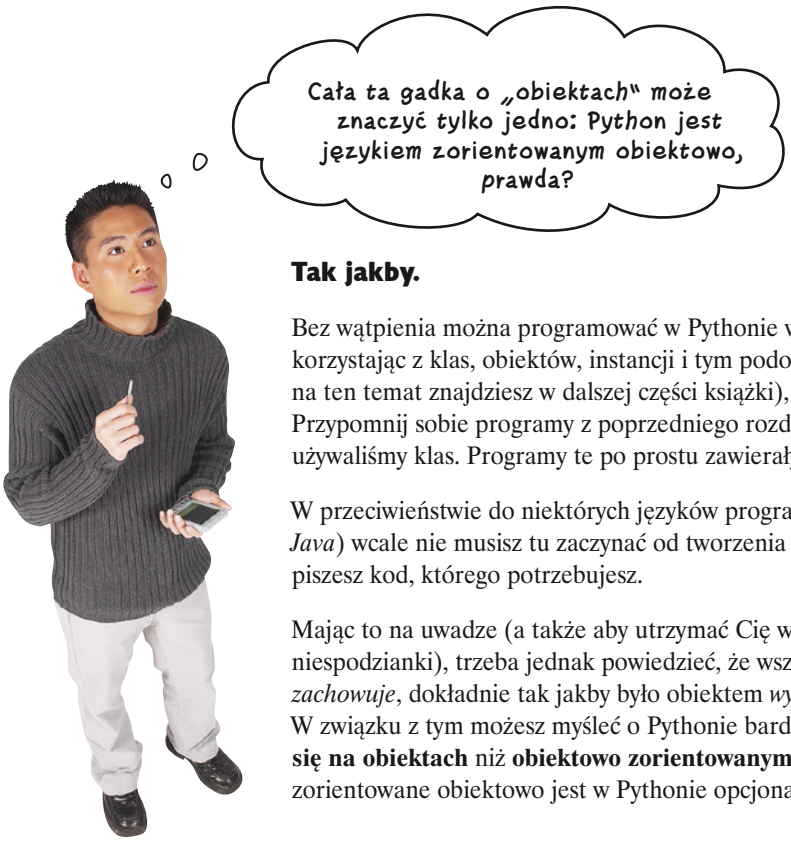
W języku Python wszystko jest obiektem. W praktyce oznacza to tyle, że liczby, łańcuchy znakowe, funkcje, moduły — dosłownie *wszystko* — stanowią obiekty. Bezpośrednią konsekwencją tego stanu rzeczy jest fakt, że wszystkie obiekty mogą być przypisywane do zmiennych. Ma to dość interesujące skutki, które zaczniesz poznawać już na następnej stronie.

W Pythonie wszystko jest obiektem, a każdy obiekt można przypisać do zmiennej.

„Wszystko jest obiektem"

W języku Python dowolny obiekt może być dynamicznie przypisany do dowolnej zmiennej. Aż prosi się tu, żeby zapytać: *czym w Pythonie jest obiekt?* Odpowiedź brzmi: **wszystko jest tu obiektem**.

Wszystkie wartości są w Pythonie obiektami, mimo że — na pierwszy rzut oka — „Podaj jajko!" jest łańcuchem znakowym, a 42 — liczbą. Dla programistów Pythona „Podaj jajko!" to *obiekt tekstowy*, 42 zaś to *obiekt liczbowy*. Podobnie jak w innych językach programowania, obiekty mogą mieć **stan** (atrybuty lub wartości) oraz **zachowanie** (metody).



Tak jakby.

Bez wątpienia można programować w Pythonie w sposób zorientowany obiektowo, korzystając z klas, obiektów, instancji i tym podobnych rzeczy (więcej informacji na ten temat znajdziesz w dalszej części książki), ale wcale nie musisz tego robić. Przypomnij sobie programy z poprzedniego rozdziału. W żadnym z nich nie używaliśmy klas. Programy te po prostu zawierały kod i działały całkiem dobrze.

W przeciwieństwie do niektórych języków programowania (zwłaszcza do języka *Java*) wcale nie musisz tu zaczynać od tworzenia jakiejś klasy, lecz po prostu piszesz kod, którego potrzebujesz.

Mając to na uwadze (a także aby utrzymać Cię w gotowości na wszelkie niespodzianki), trzeba jednak powiedzieć, że wszystko w Pythonie jakoś się *zachowuje*, dokładnie tak jakby było obiektem *wywiedzionym z* jakiejś klasy. W związku z tym możesz myśleć o Pythonie bardziej jako o języku **opierającym się na obiektach** niż **obiektowo zorientowanym**, co oznacza, że programowanie zorientowane obiektowo jest w Pythonie opcjonalne, a nie obowiązkowe.

Ale... co to wszystko właściwie znaczy?

Jako że wszystko jest w Pythonie obiektem, każda „rzecz" może zostać przypisana do każdej zmiennej, a zmiennym można przypisywać *cokolwiek* (bez względu na to, czym ta rzecz naprawdę jest: łańcuchem znakowym, funkcją, widżetem... czy też dosłownie dowolnym innym obiektem). Odłóż to teraz gdzieś głęboko na spód swojej pamięci; w tej książce będziemy wracać do tego tematu jeszcze wielokrotnie.

O przechowywaniu pojedynczych danych w zmiennych nie da się właściwie powiedzieć wiele więcej. Przyjrzymy się teraz wbudowanym w Pythona możliwościom w zakresie przechowywania **kolekcji** wartości.

Poznaj cztery wbudowane struktury danych

Python został wyposażony w **cztery** wbudowane *struktury danych*, z których możesz korzystać do przechowywania dowolnych *kolekcji* obiektów; są to: **lista**, **słownik**, **krotka** oraz **zbiór**.

Zauważ, że pisząc „wbudowane”, mamy na myśli to, że listy, krotki, słowniki i zbiory można w każdej chwili zastosować w swoim kodzie i *nie trzeba ich importować przed użyciem*, ponieważ każda z tych struktur danych stanowi integralną część samego języka.

Na kilku kolejnych stronach zaprezentujemy przegląd cech i możliwości każdej z tych wbudowanych struktur danych. Być może poczujesz pokusę, aby pominąć ten fragment książki, ale prosimy Cię, abyś się jej oparł.

Jeśli wydaje Ci się, że masz już całkiem dobre pojęcie o tym, czym jest **lista** (ang. *list*), pomyśl raz jeszcze. Lista w Pythonie jest bardziej podobna do czegoś, o czym mógłbyś myśleć jako o *tablicy*, a nie do **powiązanej listy**, która zwykle przychodzi programistom na myśl, gdy słyszą słowo „lista”. (Jeśli jesteś szczęściarzem i wiesz, co to lista powiązana, usiądź i ciesz się tą świadomością).

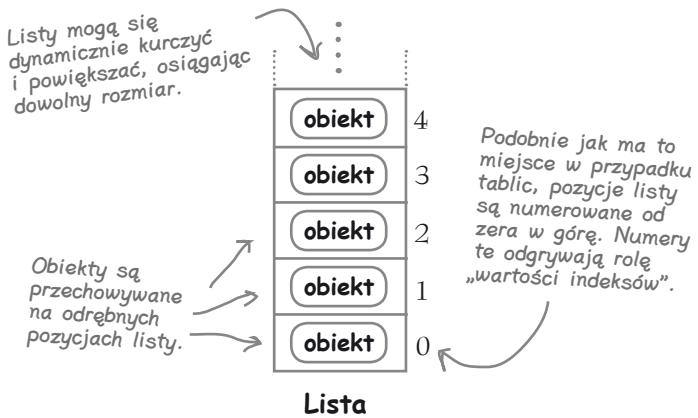
Lista jest pierwszym z dwóch rodzajów uporządkowanych kolekcji wśród struktur danych dostępnych w języku Python.

1. **Lista: uporządkowana, zmienna kolekcja obiektów.**
   Lista w Pythonie bardzo przypomina pojęciowo **tablicę** znaną z innych języków programowania w tym sensie, że stanowi indeksowaną kolekcję związanych ze sobą obiektów, w której kolejne pozycje są numerowane od zera w górę.

   W przeciwieństwie jednak do tablic używanych w innych językach listy w Pythonie są **dynamiczne**, co oznacza, że mogą się rozrastać (i zmniejszać) na życzenie. Nie ma tu potrzeby wstępnego deklarowania rozmiaru listy przed wykorzystaniem jej do przechowywania jakichkolwiek obiektów.

   Listy są również heterogeniczne w tym sensie, że nie musisz wstępnie deklarować typu obiektów, które będą w nich przechowywane — w ramach jednej listy możesz dowolnie mieszać obiekty różnych typów.

   Listy są **zmienne** (czy też **„zmienialne”**), co oznacza, że listę możesz zmienić w dowolnym momencie, dodając, usuwając lub zmieniając przechowywane w niej obiekty.



Lista przypomina tablicę — obiekty, które przechowuje, są uporządkowane na kolejnych pozycjach.

Uporządkowane kolekcje są zmienne lub niezmienne

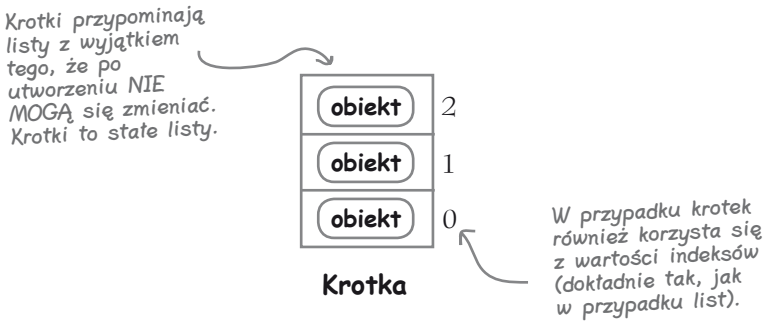
Lista w Pyhonie jest przykładem **zmiennej** (**modyfikowalnej**) struktury danych w tym sensie, że może być zmieniana (czy też modyfikowana) w czasie wykonania programu. Lista może się rozrastać lub kurczyć, gdy w miarę potrzeby dodajesz do niej lub usuwasz z niej obiekty. Możesz również zmieniać dowolny obiekt przechowywany na każdej pozycji. Już wkrótce bardzo szczegółowo omówimy ten temat, jako że cała dalsza część rozdziału będzie poświęcona dokładnemu opisowi sposobów używania list.

Gdy mamy do czynienia z przypominającą listę kolekcją, która jest **niezmienna** (a więc taką, która nie może być modyfikowana), kolekcja jest określana mianem **krotki** (ang. *tuple*).

2 **Krotka: uporządkowana, niezmienna kolekcja obiektów.**
Krotka jest niezmienną listą. Oznacza to, że gdy raz przypiszesz obiekty do krotki, niezależnie od okoliczności nie może ona zostać zmodyfikowana.

Krotki warto zwykle traktować jak stałe listy.

Większość programistów początkujących w języku Python w zakłopotaniu drapie się w głowę, gdy po raz pierwszy w swojej karierze napotyka krotki, ponieważ niezwykle trudno zrozumieć im przeznaczenie tych struktur danych. W końcu, jaki sens ma używanie listy, której nie da się zmienić? Okazuje się jednak, że istnieje mnóstwo zastosowań, w których chcesz mieć pewność, że używane przez Ciebie obiekty nie mogą zostać zmodyfikowane przez Twój kod (lub też kod napisany przez kogokolwiek innego). Do omawiania krotek wrócimy w następnym rozdziale (a także w dalszej części tej książki), gdzie przedstawimy nieco więcej szczegółów dotyczących tych struktur danych, jak również sposoby ich używania.



Krotka to niemodyfikowalna lista.

Listy i krotki sprawdzają się doskonale, gdy chcesz przedstawiać dane w sposób uporządkowany (jak ma to na przykład miejsce w planie podróży, gdy kolejność punktów docelowych *jest* istotna). Czasami jednak porządek prezentacji danych *nie jest* ważny. Możesz na przykład chcieć przechowywać jakieś informacje na temat użytkownika (takie jak jego *identyfikator* i *hasło*), ale może Cię zupełnie nie obchodzić, w jakiej kolejności są one zapisane (a interesować tylko to, że w ogóle są). W przypadku tego rodzaju danych konieczne jest zastosowanie struktur alternatywnych dla pythonowych list i krotek.

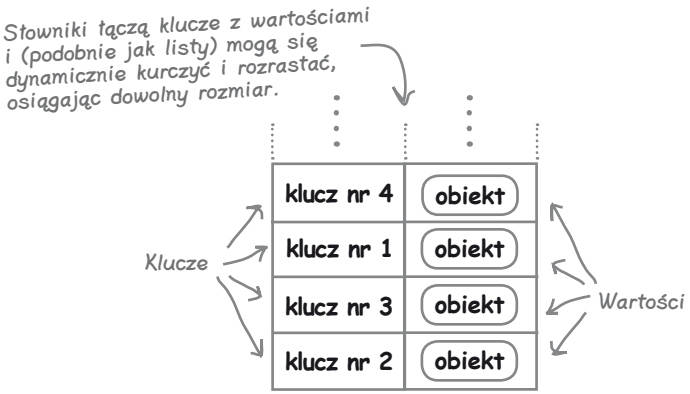
Nieuporządkowana struktura danych: słownik

Jeśli nie zależy Ci na przechowywaniu danych w określonej kolejności, ale liczy się ich struktura, to wiedz, że Python oferuje dwa rodzaje nieuporządkowanych struktur danych: **słownik** i **zbiór**. Przyjrzyjmy się po kolei każdemu z nich, zaczynając od pythonowego słownika.

3 **Słownik: nieuporządkowany zbiór par klucz-wartość.**
Jeśli masz już jakieś doświadczenie programistyczne, niewykluczone, że wiesz już, czym jest **słownik** (ang. *dictionary*), możesz jednak znać go pod inną nazwą, taką jak tablica asocjacyjna (lub też skojarzeniowa), mapa, tablica symboli czy tablica mieszająca (ang. *hash*).

Podobnie jak ma to miejsce w przypadku tych struktur danych w innych językach programowania, słownik w Pythonie umożliwia przechowywanie kolekcji par klucz-wartość. Z każdym unikatowym **kluczem** w słowniku związana jest **wartość**, a słowniki mogą przechowywać dowolną liczbę takich par. Wartościami związanymi z kluczami mogą być dowolne obiekty.

Słowniki są nieuporządkowane i zmienne. Pythonowy słownik możesz traktować jak dwukolumnową, wielowierszową strukturę danych. Tak jak listy, słowniki mogą się rozrastać (i kurczyć) w zależności od potrzeby.



Słownik

Słownik przechowuje pary klucz-wartość.

Coś, na co należy zwrócić uwagę, gdy korzysta się ze słownika, to fakt, że nie można polegać na wewnętrznym porządku elementów używanym przez interpreter. W szczególności: interpreter nie zachowuje kolejności, w której wprowadza się do słownika pary klucz-wartość, kolejność ta nie ma też żadnego znaczenia (dla Pythona). Może to stanowić nie lada zagwozdkę dla programistów, którzy po raz pierwszy stykają się z tą właściwością pythonowych słowników, dlatego ostrzegamy Cię o niej już teraz. Dzięki temu, gdy ponownie — i bardziej szczegółowo — zajmiemy się słownikami w następnym rozdziale, nie doznasz zbyt wielkiego szoku. Zachowaj jednak spokój: jeśli jest to konieczne, dane przechowywane w słownikach da się wyświetlać w określonym porządku. W kolejnym rozdziale zajmiemy się również opisem sposobu umożliwiającego zrobienie tego w Pythonie.

Struktura danych, która eliminuje duplikaty: zbiór

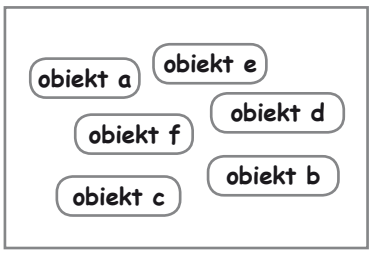
Ostatnią wbudowaną strukturą danych jest **zbiór** (ang. *set*), który bardzo się przydaje, gdy chcesz szybko usunąć duplikaty z jakiejkolwiek innej kolekcji. Nie martw się, jeśli pojęcie to przypomina Ci lekcje matematyki w szkole średniej i nadal wywołuje zimne poty. Pythonowa implementacja zbiorów może być wykorzystywana w bardzo wielu miejscach.

4 **Zbiór: nieuporządkowany zbiór unikatowych obiektów.**
Zbiór jest w Pythonie doskonałą strukturą danych do zapamiętywania kolekcji związanych ze sobą obiektów, zapewniającą przy tym, że żaden z tych obiektów się nie powtarza.

Dodatkową zaletą zbiorów w Pythonie jest to, że umożliwiają one przeprowadzanie typowych mnogościowych działań na zbiorach, takich jak suma, iloczyn i różnica zbiorów (zaletę tę docenią zwłaszcza osoby o upodobaniach matematycznych, a już szczególnie te, które szczerym uczuciem darzą teorię zbiorów).

Zbiory, podobnie jak listy i słowniki, mogą się rozrastać (i kurczyć) zgodnie z potrzebami. Podobnie jak słowniki, zbiory są nieuporządkowane, dlatego nie możesz przyjmować żadnych założeń odnośnie kolejności ułożenia obiektów w Twoim zbiorze. Zbiory, tak jak krotki i słowniki, ujrzysz w akcji już w następnym rozdziale.

Traktuj zbiór jak kolekcję nieuporządkowanych, unikatowych elementów, w której niedopuszczalne jest występowanie duplikatów.



Zbiór

Zbiór nie dopuszcza występowania duplikatów obiektów.

Praktyczna zasada 80/20 dotycząca struktur danych

Cztery wbudowane struktury danych są bardzo przydatne, ale nie zaspokajają każdej potencjalnej potrzeby związanej z danymi, choć pokrywają całkiem sporą ich część. Mamy tu do czynienia z klasyczną historią technologii opracowanej w taki sposób, aby mieć ogólne zastosowanie: możliwości wbudowanych struktur sprawdzają się świetnie w około 80% przypadków, podczas gdy w pozostałych przypadkach, a jest ich około 20%, trzeba zainwestować nieco więcej pracy. W dalszej części niniejszej książki dowiesz się, jak zwiększyć możliwości Pythona, aby sprostać wszelkim specyficznym, stawianym danym wymaganiom, z którymi możesz się spotkać. Teraz jednak, jak również w pozostałej części tego rozdziału i w całym następnym, skoncentrujemy się na wspomnianych 80% Twoich potrzeb związanych z danymi.

Resztę tego rozdziału poświęcimy omówieniu sposobów posługiwania się pierwszą z naszych czterech wbudowanych struktur danych, czyli **listą**. Pozostałe trzy struktury, a więc **słownik**, **zbiór** i **krotkę**, poznasz bliżej w kolejnym rozdziale tej książki.

Lista to uporządkowana kolekcja obiektów

Gdy posługujesz się zestawem związanych ze sobą obiektów i chcesz umieścić je gdzieś w swoim kodzie, pomyśl o **liście**. Wyobraź sobie na przykład, że masz codzienne odczyty temperatury z całego miesiąca; przechowywanie tych odczytów na liście wydaje się jak najbardziej sensowne.

Podczas gdy w innych językach programowania tablice mają raczej charakter homogeniczny — w tym znaczeniu, że możesz mieć albo tablicę liczb całkowitych, albo tablicę łańcuchów znakowych, albo tablicę odczytów temperatury — **lista** w Pythonie jest znacznie bardziej tolerancyjna w kwestii typów danych. Możesz więc korzystać z listy *obiektów*, z których każdy jest innego typu. Poza tym, że listy mają charakter **heterogeniczny**, są też **dynamiczne**, co oznacza, że mogą się rozrastać i kurczyć odpowiednio do potrzeb.

Zanim dowiesz się, jak posługiwać się listami, poświęćmy trochę czasu na naukę umiejętności dostrzegania list w kodzie Pythona.

Jak dostrzec listę w kodzie

Listy są zawsze ujęte w **nawiasy prostokątne**, a obiekty, które się na nich znajdują, zawsze są rozdzielone **przecinkami**.

Przypomnij sobie przedstawioną w poprzednim rozdziale listę `odds`, która zawierała liczby nieparzyste z przedziału od 0 do 60, tak jak zostało to pokazane poniżej.

To jest lista liczb nieparzystych.

```
                ...
odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]
                ...
```

Listę zaczyna otwierający nawias kwadratowy.

Poszczególne wartości danych (znane również jako „obiekty") są oddzielone od siebie przecinkami.

Listę kończy zamykający nawias kwadratowy.

Listę, która jest tworzona poprzez przypisanie bezpośrednio w kodzie odpowiednich obiektów do nowej listy (tak jak pokazano to powyżej), programiści Pythona określają mianem **listy literalnej** (lub **dosłownej**). Lista tego rodzaju jest tworzona *i* wypełniana za jednym zamachem.

Listy można tworzyć literalnie lub „powiększać" w kodzie.

Innym sposobem jest utworzenie listy, a następnie wypełnianie jej (aby „wzrastała") w kodzie poprzez dodawanie obiektów wraz z tym, jak jest on wykonywany. Przykład zastosowania tej metody poznasz w dalszej części tego rozdziału.

Spójrzmy na kilka przykładów list literalnych.

Literalne tworzenie list

W naszym pierwszym przykładzie utworzymy pustą listę poprzez przypisanie `[]` do zmiennej o nazwie `prices`.

```
prices = []
```

Nazwa zmiennej znajduje się na lewo od operatora przypisania...

...a „literalna lista" na prawo od niego. W tym przypadku nasza lista jest pusta.

Poniżej znajduje się lista temperatur wyrażonych w stopniach Celsjusza, czyli lista wartości zmiennoprzecinkowych.

```
temps = [ 0.0, 100.0, -17.78, 27.5, 37.78, 7.39 ]
```

Obiekty (w tym przypadku jakieś wartości zmiennoprzecinkowe) są rozdzielone przecinkami i ograniczone nawiasami kwadratowymi — mamy tu więc listę.

A może by tak utworzyć listę najbardziej znanych słów w historii programowania komputerów? Sprawdź je poniżej.

```
words = [ 'witaj', 'świecie' ]
```

Lista obiektów tekstowych

Poniżej znajdziesz listę z danymi na temat samochodu. Zauważ, że nie ma tu najmniejszego problemu z przechowywaniem danych różnych typów na tej samej liście. Przypomnij sobie, że lista jest „kolekcją związanych ze sobą obiektów". W tym przykładzie widoczne są dwa łańcuchy znakowe, jedna wartość zmiennoprzecinkowa i jedna wartość całkowita — *wszystkie* te dane są obiektami Pythona, dlatego można je przechowywać na liście, jeśli jest taka potrzeba.

```
car_details = [ 'Toyota', 'RAV4', 2.2, 60807 ]
```

Lista obiektów o różnych typach

W przypadku dwóch ostatnich przykładów list literalnych, które tu przedstawimy, wykorzystany został ten sam fakt, który pozwolił nam utworzyć poprzednią listę, a mianowicie to, że wszystko jest w Pythonie obiektem. Podobnie jak łańcuchy znakowe, wartości zmiennoprzecinkowe czy wartości całkowite, *listy również są obiektami*. Oto przykład listy składającej się z list obiektów.

```
everything = [ prices, temps, words, car_details ]
```

A tutaj znajdziesz przykład literalnej listy składającej się z literalnych list.

```
odds_and_ends = [ [ 1, 2, 3], ['a', 'b', 'c' ], [ 'Jeden',
'Dwa', 'Trzy' ] ]
```

Listy wewnątrz listy

Nie przejmuj się, jeśli te dwa ostatnie przykłady doprowadzają Cię na skraj załamania nerwowego. Z niczym tak skomplikowanym nie będziemy mieć do czynienia w najbliższym czasie.

Zaprzęganie list do pracy

Przykłady z literalnymi listami przedstawione na poprzedniej stronie pokazują, jak szybko można tworzyć i wypełniać listy w kodzie. Wystarczy, że wpiszesz odpowiednie dane i sprawę masz załatwioną.

Na stronie lub dwóch opiszemy mechanizm, który umożliwia powiększanie (lub zmniejszanie) listy w czasie działania programu. W końcu zdarza się mnóstwo sytuacji, w których nie wiesz z góry, jakie dane będziesz musiał przechowywać ani ile obiektów będzie Ci potrzebnych. W tego rodzaju przypadkach o wzrost (lub „generowanie") listy zgodnie z potrzebą musisz zadbać w swoim kodzie. Dowiesz się, jak to zrobić, już na najbliższych stronach.

Teraz wyobraź sobie, że Twoim zadaniem jest określenie, czy dane słowo zawiera wybrane samogłoski (czyli w naszym przypadku litery: *a*, *e*, *i*, *o* oraz *u*). Czy możemy skorzystać z możliwości oferowanych przez listy Pythona, aby poradzić sobie z tym zadaniem? Sprawdźmy, czy uda nam się wypracować jakieś rozwiązanie, eksperymentując w powłoce.

Korzystanie z list

Najpierw użyjemy powłoki, aby zdefiniować naszą listę poszukiwanych znaków o nazwie `vowels`, a następnie sprawdzimy, czy poszczególne litery tworzące słowo znajdują się na tej liście. Zdefiniujmy zatem tę listę.

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
```

Lista zawierająca pięć samogłosek

Mając zdefiniowaną listę `vowels`, potrzebujemy słowa, które należy sprawdzić, dlatego utwórzmy teraz zmienną o nazwie `word` i przypiszmy jej wartość `"Miliard"`.

Oto słowo do sprawdzenia

```
>>> word = "Miliard"
```

Porady maniaka

Aby uprościć nasz przykład, na liście poszukiwanych znaków umieściliśmy jedynie litery *a*, *e*, *i*, *o* oraz *u* — oczywiście nie są to wszystkie samogłoski istniejące w języku polskim. Przechowująca tę listę zmienna nosi nazwę `vowels`. W dalszej części tej książki, mówiąc o samogłoskach, będziemy mieli na myśli właśnie tę naszą skróconą listę.

Czy jeden obiekt znajduje się w drugim? Sprawdź to za pomocą operatora in

Jeśli pamiętasz programy z rozdziału 1., z pewnością przypominasz sobie, że korzystaliśmy w nich z oferowanego przez Pythona operatora `in`, aby sprawdzić, czy jeden obiekt zawierał się w drugim. Tutaj również możemy skorzystać z możliwości zapewnianych przez ten operator.

```
>>> for letter in word:
        if letter in vowels:
            print(letter)
i
i
a
```

Weź każdą literę w słowie...

...i jeśli znajduje się na liście vowels...

...wyświetl tę literę na ekranie.

Dane wyjściowe wyświetlone w wyniku wykonania tego kodu potwierdzają istnienie wskazanych samogłosek w słowie „Miliard".

Kod ten będzie dla nas stanowił podstawę kolejnych przykładów zastosowania list.

Korzystaj z edytora, gdy pracujesz nad dłuższym fragmentem kodu

Aby dowiedzieć się nieco więcej na temat tego, jak działają listy, weźmy kod przedstawiony na poprzedniej stronie i rozbudujmy go tak, aby wyświetlał każdą znalezioną samogłoskę tylko raz. W tej chwili wyświetlane są wszystkie samogłoski znalezione w słowie, a więc niektóre z nich mogą się pojawiać wielokrotnie w przypadku, gdy sprawdzany wyraz zawiera więcej niż jedno wystąpienie danej samogłoski.

Najpierw skopiujmy kod, który przed chwilą wprowadziłeś w oknie powłoki, i wklejmy go do nowego okna edycji środowiska IDLE. Można to zrobić za pomocą polecenia menu *File/New File*. W kodzie tym będziemy robić szereg zmian, dlatego przeniesienie go do edytora jest zdecydowanie dobrym pomysłem. Warto przyjąć jako ogólną zasadę, że gdy długość kodu, z którym eksperymentujemy w powłoce, zaczyna przekraczać kilka wierszy, wygodniejszym rozwiązaniem jest skorzystanie z edytora. Nasz fragment kodu zapisz pod postacią pliku o nazwie *vowels.py*.

Kopiując kod z powłoki do edytora, **zwróć uwagę**, aby *nie* zaznaczyć znaku zachęty >>>, ponieważ jeśli to zrobisz, nie uda Ci się uruchomić swojego kodu (interpreter zgłosi błąd składni, gdy napotka ciąg znaków >>>).

Gdy już skopiujesz kod i zapiszesz go w pliku, Twoje okno środowiska IDLE powinno przypominać to, które zostało przedstawione poniżej.

Kod przykładu z listą zapisany w pliku vowels.py, widoczny w oknie edycji środowiska IDLE

vowels.py - C:/Users/lsuma/Desktop/Python/...

File Edit Format Run Options Window Help

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Miliard"
for letter in word:
    if letter in vowels:
        print(letter)
```

Ln: 7 Col: 0

Nie zapomnij: naciśnij klawisz F5, aby uruchomić swój program

Widząc swój kod w oknie edycji, naciśnij klawisz *F5* i zobacz, jak IDLE przeskakuje do zrestartowanego okna powłoki, po czym wyświetla dane wyjściowe Twojego programu.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
============ RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels.py ============
i
i
a
>>>
```

Ln: 696 Col: 4

Jak można się było spodziewać, widoczne tu dane wyjściowe są identyczne z tymi, które udało się nam otrzymać, uruchamiając kod w oknie powłoki, i które widoczne są na poprzedniej stronie. Oznacza to, że możemy ruszać dalej.

„Rozrost" listy w czasie wykonania

Nasz bieżący program *wyświetla* na ekranie każdą znalezioną samogłoskę włącznie z każdym duplikatem, jaki napotka. Aby wymienić tylko unikatowe wystąpienia samogłosek (i uniknąć wyświetlania duplikatów), musimy zapamiętywać każdą z tych, które zostaną znalezione, zanim pokażemy je na ekranie. Aby to zrobić, należy skorzystać z drugiej struktury danych.

Nie możemy użyć istniejącej listy `vowels`, ponieważ jest nam ona potrzebna do szybkiego sprawdzania, czy litera, którą właśnie przetwarzamy, jest samogłoską. Potrzebna nam jest druga lista, która na początku będzie zupełnie pusta i będzie wypełniana w czasie działania programu kolejnymi samogłoskami, które uda nam się znaleźć.

Tak jak robiliśmy to w poprzednim rozdziale, spróbujmy poeksperymentować w powłoce *przed* wprowadzeniem jakichkolwiek zmian w kodzie naszego programu. Aby utworzyć nową, pustą listę, należy wymyślić jakąś nazwę zmiennej, a następnie przypisać jej pustą listę. Naszej drugiej liście nadajmy nazwę `found`. Poniżej przypisujemy zmiennej `found` pustą listę `[]`, a następnie korzystamy z wbudowanej funkcji Pythona o nazwie `len`, aby sprawdzić, ile obiektów należy do kolekcji.

```
>>> found = []
>>> len(found)
0
```

Pusta lista...

...co do której, dzięki interpreterowi (i funkcji len), mamy pewność, że nie zawiera żadnych obiektów.

Wbudowana funkcja len wyświetla informacje o rozmiarze obiektu.

Listy oferują zestaw wbudowanych **metod**, których można używać do manipulowania należącymi do nich obiektami. Aby wywołać metodę, skorzystaj ze *składni z notacją kropkową*, a więc dodaj kropkę na końcu nazwy listy, a po niej — odpowiednie wywołanie. Z wieloma metodami zapoznasz się w dalszej części tego rozdziału. Teraz skorzystajmy z metody `append`, aby dodać obiekt na końcu pustej listy, którą przed chwilą utworzyliśmy.

```
>>> found.append('a')
>>> len(found)
1
>>> found
['a']
```

Dodawanie elementu do istniejącej listy w czasie wykonania programu za pomocą metody append.

Długość listy właśnie się zwiększyła.

Wyświetlenie zawartości listy przez powłokę stanowi potwierdzenie, że obiekt jest teraz częścią listy.

Ponowne wywołania metody `append` powodują dodanie kolejnych obiektów na końcu listy.

```
>>> found.append('e')
>>> found.append('i')
>>> found.append('o')
>>> len(found)
4
>>> found
['a', 'e', 'i', 'o']
```

Kolejne elementy zostają dodane w czasie wykonania.

Ponownie korzystamy z powłoki, aby potwierdzić, że wszystko jest w porządku.

Listy oferują sporo wbudowanych metod.

Sprawdźmy teraz, co trzeba zrobić, aby przekonać się, czy lista zawiera obiekt.

Sprawdzanie przynależności za pomocą operatora in

Wiemy już, jak to zrobić. Przypomnij sobie przykład ze słowem „Miliard” zamieszczony na jednej z ostatnich stron, a także kod znany Ci z poprzedniego rozdziału zapisany w pliku *odds.py*. Za pomocą tego kodu sprawdzaliśmy, czy obliczona wartość minuty znajdowała się na liście `odds`.

Operator in umożliwia sprawdzenie przynależności.

```
...
if right_this_minute in odds:
    print("Ta minuta wydaje się dość nieparzysta.")
...
```

Czy obiekt jest w środku, czy też nie jest?

Tak jak można korzystać z operatora `in`, aby sprawdzić, czy obiekt znajduje się w kolekcji, tak da się również sprawdzić, czy obiekt *nie występuje w kolekcji*; robi się to za pomocą kombinacji operatorów `not in`.

Zastosowanie operatorów `not in` umożliwi Ci dodanie elementu do istniejącej listy *tylko* wówczas, gdy będziesz wiedział, że obiekt, który ma zostać dodany, nie jest jeszcze częścią listy.

```
>>> if 'u' not in found:
        found.append('u')
>>> found
['a', 'e', 'i' 'o', 'u']
>>>
>>> if 'u' not in found:
        found.append('u')
>>> found
['a', 'e', 'i' 'o', 'u']
```

Pierwsze wywołanie funkcji append działa, ponieważ litera „u” nie występuje obecnie na liście found (jak widziałeś na poprzedniej stronie, lista ta miała wcześniej następującą zawartość: ['a', 'e', 'i', 'o']).

To następne wywołanie funkcji append nie zostaje wykonane, ponieważ „u” znajduje się już na liście found, a zatem nie musi być dodawane ponownie.

Czy nie byłoby tu lepiej zastosować zbiór? Czy nie jest on sensowniejszą opcją, gdy zależy nam na uniknięciu występowania duplikatów?

Dobry pomysł. Zbiór mógłby się tu sprawdzić lepiej.

Z używaniem zbiorów zamierzamy się jednak wstrzymać aż do następnego rozdziału. Wrócimy do tego przykładu, gdy przyjdzie na to pora. Teraz chcemy skupić się na prezentacji sposobu generowania listy za pomocą funkcji `append` w czasie wykonania programu.

Czas na aktualizację kodu

Teraz, gdy wiemy już o operatorach `not in` oraz funkcji `append`, możemy zmodyfikować nasz kod z przeświadczeniem, że wszystko powinno pójść dobrze. Poniżej znajdziesz oryginalny kod zapisany w pliku *vowels.py*.

Oryginalny kod zapisany w pliku vowels.py

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Miliard"
for letter in word:
    if letter in vowels:
        print(letter)
```

Ten kod odpowiada za wyświetlanie samogłosek należących do słowa przechowywanego w zmiennej word, gdy są one znajdowane.

Zapisz kopię tego kodu w pliku *vowels2.py*, abyś mógł spokojnie zmieniać tę nową wersję, pozostawiając jednocześnie oryginalny kod nienaruszony.

Na początek powinniśmy tu dodać kod odpowiedzialny za tworzenie pustej listy `found`. Następnie potrzebujemy trochę więcej kodu, aby zapełnić tę listę w czasie wykonywania programu. Jako że nie będziemy już wyświetlać samogłosek w momencie ich znalezienia, potrzebna nam jest kolejna pętla `for`, aby przetworzyć litery należące do listy `found`; pętla ta powinna się wykonać dopiero *po* pierwszej pętli (zauważ poniżej, że wcięcia obydwu tych pętli są *wyrównane*). Nowy kod, który należy dodać, został zaznaczony za pomocą tła.

To zawartość pliku vowels2.py.

Zaczynamy od pustej listy.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Miliard"
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Fragment kodu, w którym podejmowana jest decyzja, czy lista znalezionych samogłosek ma zostać zaktualizowana.

Gdy pierwsza pętla for kończy działanie, ta druga rusza do pracy i wyświetla samogłoski znalezione w łańcuchu przechowywanym w zmiennej word.

Zróbmy w tym kodzie jeszcze jedną, ostatnią już zmianę. Modyfikacja dotyczy wiersza, w którym zmiennej `word` przypisywana jest wartość `"Miliard"`. Sprawmy, aby kod ten był bardziej *ogólny* i bardziej *interaktywny*.

Zamiana następującego wiersza kodu:

```
word = "Miliard"
```

na ten:

```
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
```

spowoduje, że interpreter *poprosi* użytkownika programu o podanie słowa, w którym chcemy znaleźć samogłoski. Funkcja `input` to kolejny element składowy całego wbudowanego dobrodziejstwa, które zapewnia nam język Python.

Zrób to sam!

Dokonaj zmiany, tak jak zostało to opisane obok, a następnie zapisz swój zaktualizowany kod w pliku *vowels3.py*.

Jazda próbna

Jeśli dokonałeś już zmiany przedstawionej na końcu poprzedniej strony i zapisałeś tę najnowszą wersję swojego programu pod postacią pliku *vowels3.py*, sprawdź, korzystając ze środowiska IDLE, jak sprawuje się ten kod w praktyce. Pamiętaj, że aby uruchomić program wielokrotnie, musisz za każdym razem wrócić do okna edycji IDLE *przed* naciśnięciem klawisza *F5*.

Oto nasza wersja pliku vowels3.py z zastosowaną zmianą źródła pochodzenia słowa do sprawdzenia.

vowels3.py - C:/Users/lsuma/Desktop/Python/R02/vowels3.py (3.6.0)

File Edit Format Run Options Window Help

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

A oto wyniki naszych testów...

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Miliard
i
a
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Mienie
i
e
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Mama
a
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Rym
>>>
```

Ln: 20 Col: 4

Uzyskane przez nas dane wyjściowe potwierdzają, że ten niewielki program działa zgodnie z założeniem, a nawet *zachowuje się prawidłowo* wtedy, gdy przeszukiwane słowo nie zawiera żadnych samogłosek wymienionych na liście. Jak się czułeś, uruchamiając swój program w środowisku IDLE?

Usuwanie obiektów z listy

Listy w Pythonie oferują podobne możliwości do tablic w innych językach programowania, a także wiele innych.

To, że listy mogą się dynamicznie rozrastać, gdy potrzeba więcej miejsca (dzięki metodzie `append`), zapewnia znaczny wzrost wydajności pracy. O szczegóły działania tego mechanizmu dba interpreter, podobnie jak ma to miejsce w przypadku wielu innych możliwości Pythona. Gdy lista potrzebuje więcej pamięci, interpreter dynamicznie *przydziela* jej tyle, ile trzeba. Podobnie jest, gdy lista się kurczy; interpreter dynamicznie *zwalnia* wtedy pamięć, która nie jest już liście potrzebna.

Korzystanie z list umożliwiają odpowiednie metody. Na czterech kolejnych stronach przedstawimy Ci cztery najbardziej przydatne z nich: `remove`, `pop`, `extend` oraz `insert`.

1. **Metoda remove: jako swój jedyny argument przyjmuje wartość obiektu.**
Metoda `remove` umożliwia usunięcie z listy pierwszego wystąpienia określonej wartości danej. Jeśli wartość ta zostaje znaleziona na liście, obiekt, który ją zawiera, jest usuwany z listy (a ona sama zmniejsza swój rozmiar o jeden). Jeśli wartość *nie* występuje na liście, interpreter *zgłasza błąd* (więcej na ten temat dowiesz się nieco później).

```
>>> nums = [1, 2, 3, 4]
>>> nums
[1, 2, 3, 4]
```

Tak właśnie wygląda lista nums, zanim wywołamy metodę remove.

To NIE jest wartość indeksu, lecz wartość, która ma zostać usunięta z listy.

```
>>> nums.remove(3)
>>> nums
[1, 2, 4]
```

Po wywołaniu funkcji remove obiekt zawierający wartość 3 zniknął z listy.

Wyciąganie obiektów z listy

Metoda `remove` sprawdza się doskonale, gdy znasz wartość obiektu, który chcesz usunąć z listy. Często jednak zdarza się, że chcesz usunąć obiekt spod określonego indeksu.

W tym celu język Python zapewnia metodę `pop`.

2 **Metoda pop: jako swój argument przyjmuje opcjonalną wartość indeksu.** Metoda `pop` usuwa z istniejącej listy *i zwraca* obiekt o określonej wartości indeksu. Jeśli wywołasz tę metodę bez podania tej wartości, zwracany jest i usuwany z listy ostatni znajdujący się na niej obiekt. Jeśli określisz wartość indeksu, zwracany i usuwany jest obiekt znajdujący się w tym miejscu. Jeśli lista jest pusta lub wywołujesz metodę `pop` z nieistniejącą wartością indeksu, interpreter *zgłasza błąd* (więcej na ten temat dowiesz się nieco później).

Obiekty zwracane przez metodę `pop` mogą być przypisywane do zmiennych, jeśli jest taka potrzeba, dzięki czemu zostają zachowane. Jeśli jednak nie zostaną one do niczego przypisane, po prostu znikają, a zajmowana przez nie pamięć jest odzyskiwana.

Przed wywołaniem metody pop nasza lista zawiera trzy obiekty.

Nie poinformowaliśmy metody pop, który element należy usunąć, dlatego działa ona w odniesieniu do ostatniego elementu znajdującego się na liście.

```
>>> nums.pop()
4
>>> nums
[1, 2]
```

Metoda pop zwraca usunięty obiekt; zajmowana przez niego pamięć jest zwalniana.

Po wywołaniu metody pop lista się kurczy.

To jest wartość indeksu. Zero odpowiada pierwszemu obiektowi na liście (czyli liczbie 1).

```
>>> nums.pop(0)
1
```

Tak samo jak wcześniej, metoda pop zwraca usunięty obiekt. Również tym razem pamięć zajmowana przez obiekt jest zwalniana przez interpreter.

```
>>> nums
[2]
```

W tym momencie lista nums jest już zredukowana do listy jednoelementowej.

Lista nums skurczyła się i jest teraz listą jednoelementową.

Poszerzanie listy o nowe obiekty

Wiesz już, że metoda append może być wykorzystywana do dodawania pojedynczego obiektu do istniejącej listy. Dostępne są też jednak inne metody, za pomocą których można dynamicznie dodawać dane do list.

3 **Metoda extend: jako swój jedyny argument przyjmuje listę obiektów.** Metoda extend przyjmuje drugą listę jako argument i dodaje każdy ze znajdujących się na niej obiektów do istniejącej listy, na rzecz której jest wywoływana. Metoda ta bardzo się przydaje, gdy trzeba połączyć dwie listy.

Tak wygląda w tej chwili lista nums; jest to lista jednoelementowa.

Tutaj podaj listę obiektów, które mają zostać dołączone do istniejącej listy.

```
>>> nums.extend([3, 4])
[2, 3, 4]
```

Rozbudowaliśmy listę nums, biorąc każdy z obiektów znajdujących się na podanej liście i dołączając go do listy nums.

Podanie w tym miejscu pustej listy jest poprawne, choć nieco niemądre (ponieważ do końca istniejącej listy nie dodajesz w ten sposób żadnych elementów). Skorzystanie tu z wywołania append([]) miałoby nieco więcej sensu, ponieważ do końca istniejącej listy dodałbyś obiekt będący pustą listą, jednak takie zastosowanie funkcji extend([]), jak ma to miejsce w tym przykładzie, po prostu nie powoduje wykonania żadnych działań.

```
>>> nums.extend([])
[2, 3, 4]
```

Z uwagi na to, że do rozbudowania listy nums wykorzystano listę niezawierającą żadnych obiektów, nic się tu nie zmienia.

Wstawianie obiektu na listę

Metody `append` oraz `extend` są bardzo intensywnie wykorzystywane, ale ich zastosowanie ogranicza się do dodawania obiektów na końcu (tj. po prawej stronie) istniejącej listy. Czasami będziesz z pewnością chciał dodawać je na jej początku (tj. po lewej stronie). Gdy tak właśnie będzie, powinieneś skorzystać z metody `insert`.

4 **Metoda insert: jako swoje argumenty przyjmuje wartość indeksu i obiektu.** Metoda `insert` umożliwia wstawianie obiektu do istniejącej listy *przed* wartością o podanym indeksie. Pozwala to wstawiać obiekty na początku istniejącej listy lub w jej dowolnym miejscu. Nie da się za pomocą tej metody wstawić obiektu na końcu listy, ale do tego służy metoda `append`.

Oto jak wyglądała lista nums po całej tej rozbudowie, która odbywała się na poprzedniej stronie.

```
>>> nums.insert(0, 1)
>>> nums
[1, 2, 3, 4]
```

Wartość (czyli „obiekt"), która ma zostać wstawiona.

Indeks obiektu, PRZED którym ma zostać wstawiony nowy element.

Wróciliśmy do stanu, od którego zaczynaliśmy nasze eksperymenty.

Po całym tym usuwaniu, wyciąganiu, poszerzaniu i wstawianiu mamy dokładnie tę samą listę, od której zaczynaliśmy kilka stron wcześniej, czyli: `[1, 2, 3, 4]`.

Zwróć uwagę na to, że za pomocą metody `insert` można umieścić obiekt w dowolnym miejscu istniejącej listy. W przedstawionym powyżej przykładzie postanowiliśmy dodać obiekt (liczbę 1) na początku naszej listy, ale równie łatwo moglibyśmy tu podać jakikolwiek inny numer miejsca *na* liście, na które wstawiony miałby być ten element. Spójrz na ostatni przykład, który został przedstawiony poniżej. Łańcuch znakowy zostaje w nim — tak dla zabawy — dodany w środku listy liczb. Można to było zrobić, korzystając z wartości 2 jako pierwszego argumentu wywołania metody `insert`.

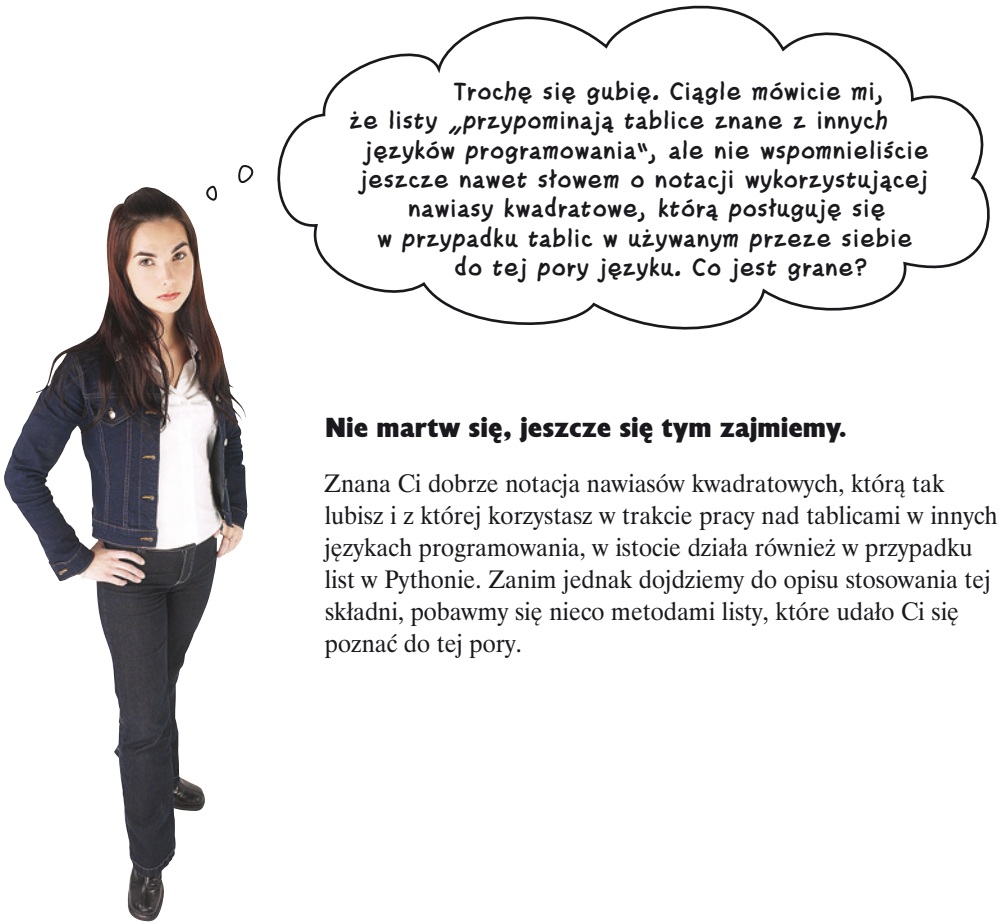
Pierwszy argument metody insert wskazuje indeks wartości, PRZED którą ma zostać wstawiony obiekt.

```
>>> nums.insert(2, "dwa-i-pół")
>>> nums
[1, 2, 'dwa-i-pół', 3, 4]
```

I oto ona: ostateczna wersja listy nums zawierająca pięć obiektów (cztery liczby i jeden łańcuch znakowy).

Nabierzmy teraz trochę doświadczenia w używaniu tych metod listy.

Co z używaniem nawiasów kwadratowych?



Nie martw się, jeszcze się tym zajmiemy.

Znana Ci dobrze notacja nawiasów kwadratowych, którą tak lubisz i z której korzystasz w trakcie pracy nad tablicami w innych językach programowania, w istocie działa również w przypadku list w Pythonie. Zanim jednak dojdziemy do opisu stosowania tej składni, pobawmy się nieco metodami listy, które udało Ci się poznać do tej pory.

Nie istnieją głupie pytania

P: Jak mam zdobyć więcej informacji o tych i innych metodach listy?

O: Wystarczy, że poprosisz o pomoc. W tym celu po znaku zachęty `>>>` napisz `help(list)`, aby uzyskać dostęp do dokumentacji Pythona (która dostarczy Ci kilku stron informacji na ten temat), lub `help(list.append)`, aby poprosić o dokumentację dotyczącą wyłącznie metody `append`. Zastąp `append` nazwą dowolnej innej metody listy, aby uzyskać pomoc na jej temat.

Zaostrz ołówek

Czas na wyzwanie.

Zanim zabierzesz się za cokolwiek innego, weź widocznych poniżej siedem wierszy kodu i wpisz je w nowym oknie edycji środowiska IDLE. Zapisz ten kod pod postacią pliku *eggs.py*, po czym go uruchom (za pomocą klawisza *F5*).

Przeczytaj komunikaty, które pojawią się na ekranie. Zauważ, że kod znajdujący się w czterech pierwszych wierszach odpowiada za zapisanie łańcucha znakowego (w zmiennej `phrase`) i przetworzenie go do postaci listy (zapisanej w zmiennej `plist`), a następnie wyświetlenie zawartości obydwu tych zmiennych na ekranie.

W pozostałych trzech wierszach odbywa się operacja odwrotna: dane zapisane w zmiennej `plist` są z powrotem przetwarzane do postaci łańcucha znakowego (zapisanego w zmiennej `new_phrase`), po czym zmienne `plist` oraz `new_phrase` wyświetlane są na ekranie.

Twoim zadaniem jest *przetworzenie* łańcucha znakowego `"Podaj jajko!"` w łańcuch `"od jaj"` tylko za pomocą tych metod umożliwiających przeprowadzanie działań na listach, które udało nam się Ci przedstawić do tej pory. (Za wyborem akurat tych łańcuchów nie kryje się żadne ukryte przesłanie, jedynym powodem ich pojawienia się tutaj jest fakt, że litery tworzące łańcuch `"od jaj"` wchodzą w skład łańcucha `"Podaj jajko!"`). Na tym etapie program *eggs.py* wyświetla *dwukrotnie* tylko łańcuch `"Podaj jajko!"`.

Podpowiedź: jeśli będzie trzeba wykonać jakąś operację wiele razy, skorzystaj z pętli `for`.

Zaczynamy od pewnego łańcucha znakowego.

Zmieniamy go w listę.

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)
```

Wyświetlamy na ekranie ten łańcuch oraz utworzoną z niego listę.

.....

.....

.....

.....

.....

.....

.....

.....

Tutaj dodaj swój kod odpowiedzialny za manipulacje na liście.

```
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Wyświetlamy na ekranie przetworzoną listę oraz nowy łańcuch znakowy.

W tym wierszu lista przetwarzana jest z powrotem do postaci łańcucha znakowego.

Zaostrz ołówek
Rozwiązanie

To był czas na wyzwanie.

Zanim zabrałeś się za cokolwiek innego, miałeś wziąć siedem wierszy kodu widocznych na poprzedniej stronie i wpisać je w nowym oknie edycji środowiska IDLE, zapisać otrzymany kod pod postacią pliku *eggs.py* i go uruchomić (za pomocą klawisza *F5*).

Twoim zadaniem było *przetworzenie* łańcucha znakowego `"Podaj jajko!"` w łańcuch `"od jaj"` tylko za pomocą poznanych do tej pory metod operujących na listach. Przed wprowadzeniem Twoich zmian program *eggs.py* wyświetlał *dwukrotnie* łańcuch `"Podaj jajko!"`.

Nowy łańcuch znakowy (`"od jaj"`) powinien był zostać zapisany w zmiennej o nazwie `new_phrase`.

W tym miejscu miałeś dodać swój kod odpowiedzialny za manipulacje na liście. Tutaj znajdziesz rozwiązanie, które udało się nam wypracować. Nie martw się, jeśli Twój kod znacznie różni się od naszego. Istnieje wiele sposobów wykonania niezbędnych transformacji za pomocą metod listy.

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove(" a ")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Ta niewielka pętla odpowiada za wyrzucenie czterech ostatnich obiektów z naszej listy plist. Nie ma już części „jko!".

Tu pozbywamy się litery „P" z początku listy.

Odszukujemy, a następnie usuwamy z listy literę „a".

Zamieniamy miejscami dwa obiekty znajdujące się na końcu listy, najpierw wyciągając z niej każdy z obiektów, a następnie używając ich do rozszerzenia listy. Jest to wiersz kodu, który musisz gruntownie przemyśleć. Kluczowe znaczenie ma tu fakt, że operacje wyciągania obiektów z listy odbywają się NAJPIERW (w kolejności, w jakiej występują w kodzie), a dopiero potem przeprowadzana jest operacja rozszerzenia listy.

Ten wiersz kodu odpowiada za wyciągnięcie z listy znaku spacji i ponowne wstawienie go na pozycję wskazywaną przez indeks 2. Podobnie jak miało to miejsce w poprzednim wierszu, operacja wyciągnięcia elementu odbywa się NAJPIERW, a operacja wstawienia dopiero po niej. Pamiętaj przy tym, że spacje również są znakami.

Z uwagi na to, że w rozwiązaniu tego ćwiczenia pojawiło się bardzo wiele interesujących rzeczy, kod ten zostanie szczegółowo wyjaśniony na dwóch następnych stronach.

Co się stało z listą plist?

Zatrzymajmy się na razie, aby spokojnie rozważyć, co tak naprawdę stało się z listą `plist` w czasie działania programu *eggs.py*.

W lewej części tej strony (i następnej) znajduje się kod zapisany w pliku *eggs.py*, który — podobnie jak każdy inny program w języku Python — wykonywany jest od góry do dołu. W prawej części strony umieszczona została graficzna reprezentacja stanu listy `plist` wraz z krótkimi notatkami opisującymi, co się dzieje w danej chwili. Zwróć uwagę na to, jak lista dynamicznie rozszerza się i kurczy wraz z tym, jak jest wykonywany kod.

Kod

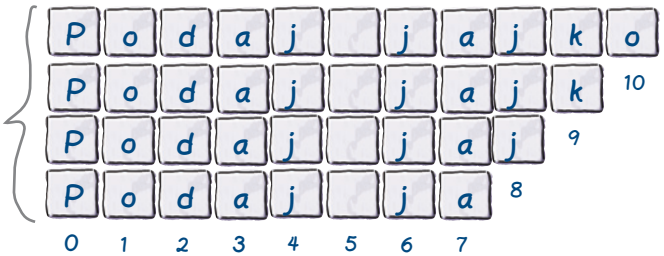
```
phrase = "Podaj jajko!"

plist = list(phrase)

print(phrase)
print(plist)

for i in range(4):
        plist.pop()

plist.pop(0)

plist.remove("a")
```

Te wywołania funkcji print powodują wyświetlenie bieżącego stanu zmiennych (przed rozpoczęciem naszych manipulacji).

Stan listy `plist`

W tym miejscu kodu lista `plist` jeszcze nie istnieje. W drugim wierszu kodu odbywa się *transformacja* łańcucha znakowego `phrase` w nową listę, która zostaje przypisana do zmiennej `plist`.

Przy każdej iteracji pętli `for` lista `plist` jest skracana o jeden obiekt aż do momentu, gdy zostaną usunięte jej ostatnie cztery elementy.



Pętla kończy swoje działanie, a lista `plist` skurczyła się do tej pory do ośmiu obiektów. Nadszedł czas na pozbycie się z niej innych niechcianych obiektów. Kolejne wywołanie metody `pop` powoduje usunięcie pierwszego elementu listy (czyli tego o indeksie 0).

Po wyciągnięciu litery `P` z początku listy następuje wywołanie metody `remove`, które powoduje usunięcie litery `a`.

Co się stało z listą plist? — ciąg dalszy

Zatrzymaliśmy się na chwilę, aby spokojnie rozważyć, co tak naprawdę stało się z listą `plist` w czasie działania programu *eggs.py*.

Analizując wykonanie kodu na poprzedniej stronie, doszliśmy do miejsca, w którym mamy sześcioelementową listę zawierającą znaki `o`, `d`, `j`, spacja, `j` oraz `a`. Kontynuujmy wykonanie kodu.

Kod

Stan listy `plist`

Poniżej widać, jak lista `plist` wygląda po wykonaniu kodu przedstawionego na poprzedniej stronie.

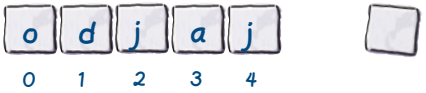
```
plist.extend([plist.pop(), plist.pop()])
```

Nowy wiersz kodu zawiera **trzy** wywołania metod: dwa wywołania metody `pop` oraz jedno metody `extend`. Wywołania metody `pop` odbywają się najpierw (w kolejności od lewej do prawej).

Wywołanie metody `extend` przyjmuje jako argument wyciągnięte przed chwilą obiekty i dodaje je do końca listy `plist`. To wywołanie metody `extend` można traktować jako wygodniejszą wersję wielokrotnego wykonania metody `append`.

Teraz została już tylko jedna rzecz do zrobienia (z listą `plist`): zamiana miejscami znajdującej się pod indeksem 2 litery `j` ze spacją umieszczoną na pozycji 3. Następny wiersz kodu zawiera **dwa** wywołania metod. Pierwszym jest wywołanie metody `pop` w celu wyciągnięcia znaku spacji.

```
plist.insert(2, plist.pop(3))
```



Drugie to wywołanie metody `insert` w celu wstawienia znaku spacji w odpowiednie miejsce (tj. przed pozycję wskazywaną przez indeks 2).

Tadam!

Ponowne przekształcenie listy plist w łańcuch znakowy.

```
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Te wywołania funkcji print powodują wyświetlenie stanu zmiennych (po przeprowadzeniu naszych manipulacji).

Listy: co wiemy

Mamy już za sobą ponad 20 stron tego rozdziału, więc zróbmy sobie krótką przerwę i sprawdźmy, czego dowiedzieliśmy się do tej pory o listach.

CELNE SPOSTRZEŻENIA

- Listy doskonale nadają się do przechowywania kolekcji związanych ze sobą obiektów. Jeśli masz pewną liczbę podobnych do siebie rzeczy, które chciałbyś traktować jak jedną całość, lista okaże się dobrym wyborem.
- Listy są podobne do tablic znanych z innych języków programowania. W przeciwieństwie jednak do tablic (które przeważnie mają z góry ustalony rozmiar) listy Pythona mogą się dynamicznie powiększać i kurczyć wedle potrzeby.
- W kodzie lista obiektów jest umieszczona w nawiasach kwadratowych, a poszczególne znajdujące się na niej obiekty są rozdzielone przecinkami.
- Pusta lista ma następującą postać: `[]`.
- Najszybszy sposób, aby przekonać się, czy obiekt znajduje się na liście, polega na skorzystaniu z zapewnianego przez język Python operatora `in`, który umożliwia sprawdzenie przynależności.
- Powiększanie listy w czasie wykonania programu umożliwiają przydatne metody, do których należą: `append`, `extend` oraz `insert`.
- Zmniejszanie listy w czasie wykonania programu umożliwiają metody `remove` oraz `pop`.

Jak dla mnie, to wszystko w porządku. Jednak czy jest coś, na co powinienem uważać, gdy posługuję się listami?

Tak. Ostrożność jest zawsze potrzebna.

Korzystanie z list i ich przetwarzanie w Pythonie jest zwykle bardzo wygodne, warto jednak zadbać o to, aby interpreter robił dokładnie to, na czym nam zależy.

Świetnym przykładem może tu być kopiowanie list. Czy robiąc to, kopiujesz całą listę, czy też tylko znajdujące się na niej obiekty? W zależności od odpowiedzi, której udzielisz na to pytanie, oraz od tego, co tak naprawdę zamierzasz zrobić, interpreter będzie zachowywał się inaczej. Przejdź do następnej strony, aby przekonać się, co dokładnie mamy tu na myśli.

Coś, co wygląda na kopiowanie, choć wcale nim nie jest

Gdy przychodzi do kopiowania jednej istniejącej listy do innej, aż prosi się o użycie operatora przypisania, tak jak zostało to pokazane poniżej.

```
>>> first = [1, 2, 3, 4, 5]
>>> first
[1, 2, 3, 4, 5]
>>> second = first
>>> second
[1, 2, 3, 4, 5]
```

Tworzenie nowej listy (i przypisanie do niej pięciu obiektów liczbowych)

Pięć liczb należących do listy o nazwie „first"

„Kopiowanie" istniejącej listy do nowej listy o nazwie „second

Pięć liczb należących do listy o nazwie „second"

Jak dotąd jest dobrze. Wygląda na to, że zastosowany przez nas sposób zadziałał, jako że pięć obiektów liczbowych znajdujących się na liście `first` zostało skopiowanych do listy `second`.

Czy aby na pewno? Sprawdźmy, co się stanie, gdy za pomocą metody `append` dołączymy nową liczbę do listy `second`. Działanie to wydaje się zupełnie rozsądne, prowadzi jednak do pojawienia się pewnego problemu.

```
>>> second.append(6)
>>> second
[1, 2, 3, 4, 5, 6]
```

To wydaje się być w porządku, ale nie jest.

Ponownie: jak dotąd wszystko jest w porządku. W rzeczywistości jednak mamy tu pewien **bug**. Spójrz, co się dzieje, gdy poprosimy powłokę o wyświetlenie zawartości listy `first`. Okazuje się, że nowy obiekt został dołączony również do niej!

```
>>> first
[1, 2, 3, 4, 5, 6]
```

Ups! Nowy obiekt został dołączony również do listy first.

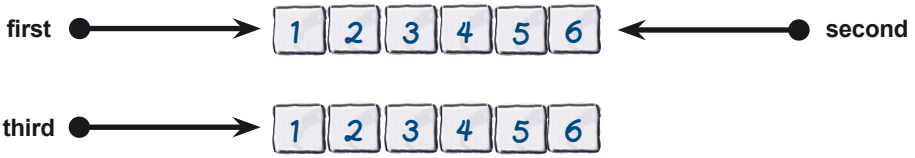
Problem polega tu na tym, że zarówno zmienna `first`, jak i zmienna `second` wskazują te same dane. Gdy zmienisz jedną listę, druga również ulegnie zmianie. Zdecydowanie nie o to nam chodziło.

Jak skopiować strukturę danych

Jeśli zastosowanie operatora przypisania nie jest odpowiednim sposobem kopiowania listy, co nim jest? To, z czym mieliśmy przed chwilą do czynienia, polegało na współdzieleniu **referencji** (czy też **odwołania**) do tej samej listy pomiędzy zmiennymi `first` oraz `second`.

Aby pomóc rozwiązać ten problem, listy zapewniają metodę `copy`, która robi dokładnie to, co trzeba. Spójrz, jak działa ta metoda.

```
>>> third = second.copy()
>>> third
[1, 2, 3, 4, 5, 6]
```



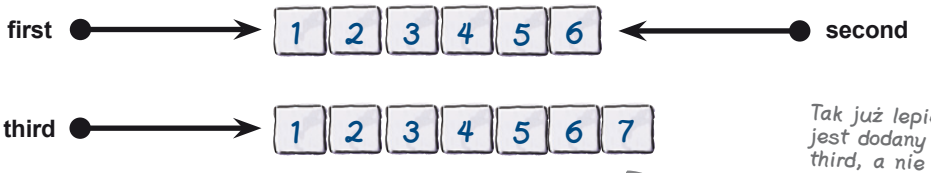
Do kopiowania list nie używaj operatora przypisania; zamiast niego korzystaj z metody copy.

Mając utworzoną listę `third` (dzięki działaniu metody `copy`), dołączmy do niej jakiś obiekt, aby zobaczyć, co się stanie.

```
>>> third.append(7)
>>> third
[1, 2, 3, 4, 5, 6, 7]
>>> second
[1, 2, 3, 4, 5, 6]
```

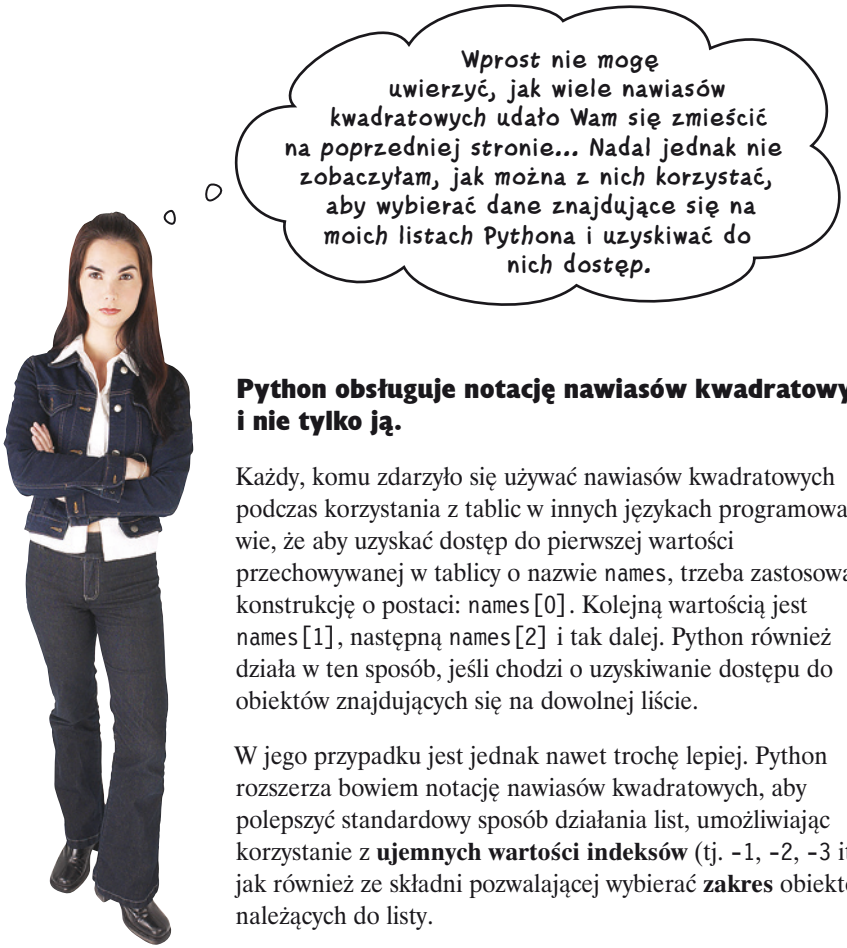
Lista third została powiększona o jeden obiekt.

Jest o wiele lepiej. Istniejąca lista pozostała niezmieniona.



Tak już lepiej — nowy obiekt jest dodany tylko do listy third, a nie do niej i dwóch pozostałych list (czyli first i second).

Nawiasy kwadratowe są wszędzie



Python obsługuje notację nawiasów kwadratowych i nie tylko ją.

Każdy, komu zdarzyło się używać nawiasów kwadratowych podczas korzystania z tablic w innych językach programowania, wie, że aby uzyskać dostęp do pierwszej wartości przechowywanej w tablicy o nazwie `names`, trzeba zastosować konstrukcję o postaci: `names[0]`. Kolejną wartością jest `names[1]`, następną `names[2]` i tak dalej. Python również działa w ten sposób, jeśli chodzi o uzyskiwanie dostępu do obiektów znajdujących się na dowolnej liście.

W jego przypadku jest jednak nawet trochę lepiej. Python rozszerza bowiem notację nawiasów kwadratowych, aby polepszyć standardowy sposób działania list, umożliwiając korzystanie z **ujemnych wartości indeksów** (tj. `-1`, `-2`, `-3` itd.), jak również ze składni pozwalającej wybierać **zakres** obiektów należących do listy.

Listy: odświeżanie tego, co już wiemy

Zanim zanurzymy się w opis sposobu, w jaki Python rozszerza notację wykorzystującą nawiasy kwadratowe, dodajmy coś do naszej listy celnych spostrzeżeń.

CELNE SPOSTRZEŻENIA

- Bądź ostrożny, kopiując listy. Jeśli chcesz, aby inna zmienna odwoływała się do istniejącej listy, skorzystaj z operatora przypisania (`=`). Jeśli jednak chcesz wykonać kopię obiektów znajdujących się na istniejącej liście, a następnie użyć ich do zainicjalizowania nowej listy, skorzystaj zamiast tego z metody `copy`.

Listy rozszerzają notację wykorzystującą nawiasy kwadratowe

Wszystkie nasze zapewnienia, że listy Pythona stanowią odpowiednik tablic w innych językach programowania, nie były tylko czczą gadaniną. Podobnie jak w innych językach, również w Pythonie indeksy komórek numeruje się od zera; analogicznie do innych języków, również tutaj w celu odwołania się do obiektów na liście stosuje się świetnie wszystkim znaną **notację nawiasów kwadratowych**.

W przeciwieństwie jednak do innych języków programowania Python umożliwia odwoływanie się do listy w odniesieniu do każdego z jej końców: za pomocą dodatnich wartości indeksów odlicza się komórki od strony lewej do prawej, podczas gdy ujemne umożliwiają odliczanie od prawej do lewej, tak jak zostało to pokazane poniżej.

Listy Pythona obsługują dodatnie wartości indeksów, które zaczynają się od 0...

...jak również ujemne wartości indeksów, które zaczynają się od –1.

Przyjrzymy się kilku przykładom zastosowania tego mechanizmu w powłoce.

```
>>> saying = "Podaj jajko!"
>>> letters = list(saying)
>>> letters
['P', 'o', 'd', "a", 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
>>> letters[0]
'P'
>>> letters[3]
"a"
>>> letters[6]
'j'
>>> letters[-1]
'!'
>>> letters[-3]
'k'
>>> letters[-6]
'j'
```

Tworzenie listy liter.

Używanie dodatnich wartości indeksów powoduje odliczanie od lewej strony do prawej...

...podczas gdy stosowanie ujemnych wartości indeksów skutkuje liczeniem od prawej do lewej.

W Pythonie bardzo łatwo jest dostać się do pierwszego i ostatniego obiektu na dowolnej liście.

```
>>> first = letters[0]
>>> last = letters[-1]
>>> first
'P'
>>> last
'!'
```

Możliwość poruszania się po liście przy użyciu ujemnych wartości indeksów często okazuje się bardzo przydatna, ponieważ listy rozrastają się i kurczą wraz z wykonywaniem kodu Pythona. Na przykład zastosowanie wartości -1 w roli indeksu zawsze gwarantuje nam dostęp do ostatniego obiektu znajdującego się na liście *bez względu na jej rozmiar*, podobnie jak użycie 0 zawsze stanowi odwołanie do jej pierwszego elementu.

Zapewniane przez Pythona rozszerzenia notacji nawiasów kwadratowych nie ograniczają się do obsługi ujemnych wartości indeksów. Listy radzą sobie również z wartościami **początku**, **końca** i **kroku**.

Listy „rozumieją” początek, koniec i krok

Pojęcia **początku**, **końca** i **kroku** zaprezentowaliśmy po raz pierwszy w poprzednim rozdziale przy okazji omawiania trzyargumentowej wersji funkcji `range`.

beersong.py - C:/Users/lsuma/Desktop/Python/R

File Edit Format Run Options Window H

```
word = "butelki"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "piwa na ś
    print(beer_num, word, "piwa.")
```

Wywołanie funkcji range przyjmuje trzy argumenty: początek, koniec i krok.

Przypomnij sobie znaczenie **początku**, **końca** i **kroku** dla określania zakresów (spróbujmy też odnieść je do list):

- **Wartość POCZĄTEK zapewnia kontrolę nad tym, GDZIE zakres się zaczyna.**
  W przypadku list wartość **początek** wskazuje początkową wartość indeksu.

- **Wartość KONIEC zapewnia kontrolę nad tym, KIEDY zakres się kończy.**
  W przypadku list wartość **koniec** wskazuje wartość indeksu, na którym należy się zatrzymać, jednak sama ta wartość **nie wschodzi w skład** definiowanego zbioru.

- **Wartość KROK zapewnia kontrolę nad tym, JAK zakres jest generowany.**
  W przypadku list wartość **krok** odnosi się do „przeskoku”, jaki należy zrobić.

Wartości początku, końca i kroku można umieszczać w nawiasach kwadratowych

W przypadku list **początek**, **koniec** i **krok** określa się *w obrębie* nawiasów kwadratowych, a odpowiednie wartości rozdziela się za pomocą znaku dwukropka (`:`).

```
letters[początek:koniec:krok]
```

Notacja wykorzystująca nawiasy kwadratowe obejmuje możliwość stosowania wartości początku, końca i kroku

Może się to wydawać nieco sprzeczne z intuicją, ale wszystkie te trzy wartości są *opcjonalne*, gdy wykorzystuje się je razem. I tak:

Gdy nie określi się **początku**, przyjmuje on standardową wartość 0.

Gdy nie określi się **końca**, przyjmuje on maksymalną wartość dopuszczalną dla listy.

Gdy nie określi się **kroku**, przyjmuje on standardową wartość 1.

Wycinki list w akcji

Korzystając z istniejącej listy `letters`, którą przedstawiliśmy Ci na jednej z poprzednich stron, możesz na różne sposoby określać wartości **początku**, **końca** i **kroku**.

Przyjrzyjmy się kilku przykładom.

Wszystkie litery

```
>>> letters
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
```

```
>>> letters[0:10:3]
['P', 'a', 'j', 'k']
```

Co trzecia litera od początku listy aż do litery znajdującej się na pozycji wskazywanej przez indeks 10 (ale bez niej samej)

```
>>> letters[3:]
['a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
```

Wszystkie znaki z wyjątkiem trzech pierwszych, które zostały pominięte

```
>>> letters[:10]
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k']
```

Wszystkie znaki od początku listy aż do litery znajdującej się na pozycji wskazywanej przez indeks 10 (ale bez niej samej)

```
>>> letters[::2]
['P', 'd', 'j', 'j', 'j', 'o']
```

Co druga litera z listy

Korzystanie z wartości początku, końca i kroku oraz *notacji wycinka* w stosunku do list daje ogromne możliwości w zakresie ich przetwarzania (nie wspominając już o tym, jak bardzo jest wygodne). Dlatego radzimy Ci poświęcić trochę czasu na dokładne zrozumienie, co się dzieje w powyższych przykładach. Spróbuj wprowadzić je w swojej powłoce po znaku zachęty >>> i nie bój się eksperymentować z tą notacją, ponieważ możesz się dzięki temu sporo nauczyć.

Nie istnieją głupie pytania

P: Tutaj co prawda nie ma takiego przypadku, ale podczas swoich eksperymentów zauważyłem, że niektóre znaki są ujęte w pojedyncze cudzysłowy, podczas gdy inne znajdują się w podwójnych. Czy istnieje jakiś standard, którego powinienem się tu trzymać?

O: Nie, nie istnieje jeden standard zapisu, a Python umożliwia Ci zastosowanie zarówno pojedynczych, jak i podwójnych cudzysłowów podczas definiowania łańcuchów znakowych o dowolnej długości, w tym również takich, które składają się z pojedynczych znaków (a więc takich jak te, które widzisz na niniejszej stronie; z technicznego punktu widzenia są one bowiem jednoznakowymi łańcuchami, a nie literami). Większość programistów korzystających z Pythona do ograniczania łańcuchów znakowych używa pojedynczych cudzysłowów (nie jest to jednak reguła, lecz wyłącznie kwestia indywidualnych upodobań). Jeśli łańcuch zawiera znak pojedynczego cudzysłowu, można zastosować podwójne cudzysłowy, aby uniknąć konieczności używania sekwencji ucieczki zawierających znak ukośnika wstecznego (\), ponieważ większość osób uważa, że łatwiej jest przeczytać ciąg `"'"` niż `'\''`. Więcej przykładów zastosowania obydwu rodzajów cudzysłowów znajdziesz na dwóch następnych stronach.

Działania na początku i końcu listy

Sprawdź w swojej powłoce przykłady zaprezentowane na tej stronie (i na stronie następnej), aby przekonać się, czy uzyskane przez Ciebie dane wyjściowe odpowiadają naszym wynikom.

Zaczynamy od przekształcenia łańcucha znakowego w listę liter.

```
>>> book = "Co gryzie Gilberta Grape'a"
>>> booklist = list(book)
>>> booklist
['C', 'o', ' ', 'g', 'r', 'y', 'z', 'i', 'e', ' ', 'G',
'i', 'l', 'b', 'e', 'r', 't', 'a', ' ', 'G', 'r', 'a',
'p', 'e', "'", 'a']
```

Zmień łańcuch znakowy w listę, a następnie wyświetl ją na ekranie.

Zauważ, że oryginalny łańcuch znakowy zawierał apostrof (znak tożsamy w tym przypadku z pojedynczym cudzysłowem). Python jest wystarczająco „sprytny", aby go wykryć i ująć w podwójny cudzysłów (zamiast w pojedynczy, jak ma to miejsce w przypadku pozostałych znaków).

Nowo utworzoną listę (noszącą tutaj nazwę `booklist`) wykorzystujemy następnie w roli bazy do wycinania fragmentów.

```
>>> booklist[0:2]
['C', 'o']
```

Wybierz z listy pierwsze dwa obiekty (litery).

```
>>> ''.join(booklist[0:2])
'Co'

>>> ''.join(booklist[-7:])
'Grape'a'
```

Zmień wskazany fragment listy w łańcuch znakowy (wiesz już, jak to zrobić, dzięki ostatnim wierszom kodu programu eggs.py). W drugim przykładzie wybieranych jest siedem ostatnich obiektów należących do listy.

Poświęć trochę czasu, aby dokładnie przestudiować tę stronę (oraz następną) i zyskać pewność, że dobrze rozumiesz, jak działa każdy z zaprezentowanych tu przykładów kodu. Koniecznie spróbuj również uruchomić każdy z nich w swoim środowisku IDLE.

Spójrz na ostatni z widocznych powyżej przykładów, aby przekonać się, że interpreter równie dobrze radzi sobie ze stosowaniem standardowych wartości w przypadku każdej z trzech wartości określających **początek**, **koniec** i **krok**.

Kroczenie przez listy

Poniżej znajdziesz jeszcze dwa przykłady zastosowania wartości **krok** w przypadku list.

W pierwszym przykładzie wybierane są wszystkie litery, począwszy od końca listy (a więc *w odwrotnej kolejności*), podczas gdy w drugim — co drugi znak. Zwróć uwagę na wpływ, jaki wartość **kroku** ma na działanie tych fragmentów kodu.

```
>>> backwards = booklist[::-1]
>>> ''.join(backwards)
"a'eparG atrebliG eizyrg oC"
```

Wygląda to na jakąś nowomowę, prawda? Ale tak naprawdę to oryginalny łańcuch tekstowy wyświetlony od końca.

```
>>> every_other = booklist[::2]
>>> ''.join(every_other)
"C rzeGlet rp'"
```

A to już jakiś całkowity bełkot! Jednak wbrew pozorom ma sens: lista every_other składa się z co drugiego obiektu (czyli znaku) listy oryginalnej, począwszy od pierwszego, a skończywszy na ostatnim. Zauważ, że wartości początku i końca nie zostały określone (dlatego są tu używane wartości standardowe).

Dwa ostatnie, zaprezentowane poniżej przykłady mają Ci pokazać, że wybieranie obiektów z listy można zacząć i zakończyć w dowolnym miejscu. Dane zwracane w wyniku tego rodzaju działania są określane mianem **wycinka** (ang. *slice*). O wycinku możesz myśleć jako o *fragmencie* istniejącej listy.

W obydwu tych przykładach wybierany jest ten sam fragment listy `booklist`, a konkretnie — słowo `'Gilberta'`. W pierwszym przypadku z odpowiednich elementów listy tworzymy po prostu zwykły łańcuch znakowy, który zostaje następnie wyświetlony na ekranie, a efektem wykonania drugiego fragmentu kodu jest wyświetlenie słowa `'Gilberta'` od końca.

```
>>> ''.join(booklist[10:18] )
'Gilberta'
```

Wycięcie słowa „Gilberta" z listy

Wycinek to fragment listy.

```
>>> ''.join(booklist[17:9:-1] )
'atrebliG'
```

Wycięcie słowa „Gilberta" z listy, ale w odwrotnej kolejności (tj. od końca)

Wycinki są wszędzie

Zastosowanie notacji wycinka nie ogranicza się wyłącznie do list. W rzeczywistości wkrótce się przekonasz, że operacji wycinania możesz używać w stosunku do dowolnej sekwencji w języku Python, a pomoże Ci w tym konstrukcja `[`*`początek`*`:`*`koniec`*`:`*`krok`*`]`.

Zaprzęganie wycinków do pracy z listami

Zapewniana przez język Python notacja wycinka to niezwykle przydatne rozszerzenie notacji nawiasów kwadratowych — jest ona wykorzystywana w bardzo wielu miejscach. W czasie lektury tej książki spotkasz się z mnóstwem zastosowań wycinków w najrozmaitszych sytuacjach.

Na razie poobserwujmy trochę pythonową notację nawiasów kwadratowych w akcji (w tym również zastosowanie wycinków). Weźmiemy teraz na warsztat przedstawiony na poprzednich stronach program *eggs.py* i zrefaktorujemy go w taki sposób, aby wykorzystywał notację nawiasów kwadratowych oraz wycinki w celu osiągnięcia tego samego efektu, który udało nam się wcześniej otrzymać za pomocą metod listy.

Zanim na poważnie zabierzemy się do pracy, warto przypomnieć sobie, co robił program *eggs.py*.

Przekształcanie „Podaj jajko!" w „od jaj"

Kod tego programu przekształca jeden łańcuch znakowy w inny, manipulując istniejącą listą przy użyciu metod listy. Zaczynając od łańcucha `"Podaj jajko!"`, kod ten generuje łańcuch `"od jaj"` za pomocą kilku przekształceń, które zostały przedstawione poniżej.

Kod programu eggs.py

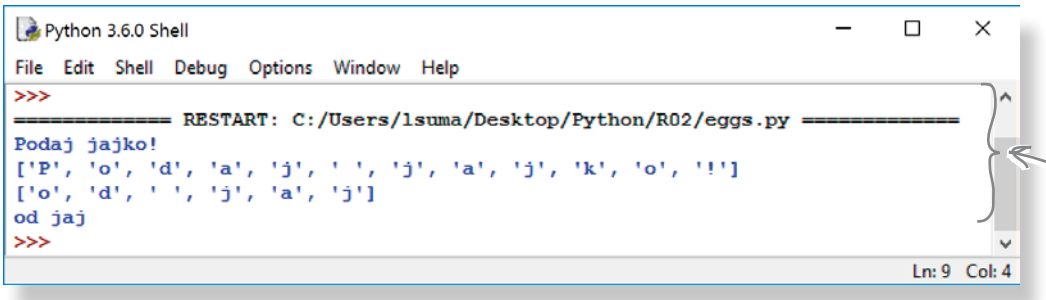
Wyświetlanie początkowego stanu łańcucha znakowego i listy

Zastosowanie kilku metod listy w celu przekształcenia listy obiektów i wykonania odpowiednich działań

Wyświetlanie końcowego stanu łańcucha znakowego

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)
for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("a")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Taki jest wynik wykonania tego programu w oknie środowiska IDLE.



Łańcuch znakowy „Podaj jajko!" został przekształcony w łańcuch „od jaj" dzięki zastosowaniu metod listy.

Zaprzęganie wycinków do pracy z listami — ciąg dalszy

Teraz nadszedł czas na zabranie się do poważnej roboty. Poniżej znajdziesz kod programu *eggs.py* z zaznaczonymi fragmentami, które należy zmienić.

Te wiersze kodu powinieneś zmienić.

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)
for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("a")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Zaostrz ołówek

Ćwiczenie polega na zastąpieniu zaznaczonego powyżej kodu nowym, który wykorzystuje możliwości oferowane przez notację nawiasów kwadratowych zapewnianą przez język Python. Zauważ, że nadal możesz używać metod listy wszędzie tam, gdzie ma to sens. Podobnie jak wcześniej, również tutaj Twoim zadaniem jest przekształcenie łańcucha znakowego `"Podaj jajko!"` w łańcuch `"od jaj"`. Dodaj odpowiedni kod w miejscach oznaczonych kropkowanymi liniami i nazwij swój nowy program *eggs2.py*.

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
print(plist)
print(new_phrase)
```

Zaostrz ołówek
Rozwiązanie

Ćwiczenie polegało na zastąpieniu kodu zaznaczonego na poprzedniej stronie nowym, który wykorzystuje możliwości oferowane przez notację nawiasów kwadratowych zapewnianą przez język Python. Zauważ, że nadal mogłeś używać metod listy wszędzie tam, gdzie miało to sens. Tak jak wcześniej, również tutaj Twoim zadaniem było przekształcenie łańcucha znakowego `"Podaj jajko!"` w łańcuch `"od jaj"`. Miałeś dodać odpowiedni kod w miejscach oznaczonych kropkowanymi liniami i nazwać swój nowy program *eggs2.py*.

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)
.....................................................
.....................................................
new_phrase = ''.join(plist[1:3])
.....................................................
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
.....................................................
.....................................................
print(plist)
print(new_phrase)
```

Zaczęliśmy od wycięcia słowa „od" z listy plist...

...aby następnie wybrać każdy kolejny znak, który był nam potrzebny, a więc: spację, „j", „a" oraz „j".

To świetne pytanie.

Niektórzy programiści spojrzą na kod programu *eggs2.py* i po porównaniu go z kodem programu *eggs.py* stwierdzą, że dwa wiersze kodu są zawsze lepsze niż siedem, zwłaszcza że dane wyjściowe obydwu tych programów są takie same. Jest to całkiem dobra miara „lepszości", jednak niezbyt użyteczna w tym przypadku.

Aby przekonać się, co mamy przez to na myśli, przyjrzyj się danym wyjściowym wygenerowanym przez obydwa programy.

Jazda próbna

Użyj środowiska IDLE, aby otworzyć kody programów *eggs.py* i *eggs2.py* w osobnych oknach edycji. Wybierz najpierw okno z programem *eggs.py*, a następnie naciśnij klawisz *F5*. Potem wybierz okno z programem *eggs2.py* i ponownie naciśnij klawisz *F5*. Porównaj wyniki działania obydwu tych programów w powłoce.

eggs.py

eggs.py - C:\Users\lsuma\Desktop\Python\R02\eggs.py (3.6.0)

File Edit Format Run Options Window Help

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
   plist.pop()
plist.pop(0)
plist.remove("a")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Ln: 17 Col: 0

eggs2.py

eggs2.py - C:/Users/lsuma/Desktop/Python/R

File Edit Format Run Options Window

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
============ RESTART: C:\Users\lsuma\Desktop\Python\R02\eggs.py ============
Podaj jajko!
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
['o', 'd', ' ', 'j', 'a', 'j']
od jaj
>>>
============ RESTART: C:/Users/lsuma/Desktop/Python/R02/eggs2.py ============
Podaj jajko!
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
od jaj
>>>
```

Ln: 27

Dane wyjściowe wygenerowane w wyniku uruchomienia programu eggs.py

Dane wyjściowe wygenerowane w wyniku uruchomienia programu eggs2.py

Zauważ, jak bardzo różnią się te dane wyjściowe.

Który jest lepszy? To zależy...

Korzystając ze środowiska IDLE, uruchomiliśmy programy *eggs.py* oraz *eggs2.py*, aby stwierdzić, który z nich jest „lepszy”.

Spójrz na przedostatni wiersz danych wyjściowych wygenerowanych przez każdy z tych programów.

To są dane wyjściowe wygenerowane przez program eggs.py...

```
>>>
============= RESTART: C:\Users\lsuma\Desktop\Python\R02\eggs.py =============
Podaj jajko!
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
['o', 'd', ' ', 'j', 'a', 'j']
od jaj
>>>
============ RESTART: C:/Users/lsuma/Desktop/Python/R02/eggs2.py ============
Podaj jajko!
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
['P', 'o', 'd', 'a', 'j', ' ', 'j', 'a', 'j', 'k', 'o', '!']
od jaj
>>>
```

...to zaś dane wygenerowane przez program eggs2.py.

Choć obydwa programy kończą swoje działanie na wyświetleniu łańcucha znakowego `"od jaj"` (zaczynając oczywiście od przetworzenia łańcucha `"Podaj jajko!"`), program *eggs2.py* nie zmienia w żaden sposób listy `plist`, podczas gdy program *eggs.py* to robi.

Warto się tu na chwilę zatrzymać, aby to rozważyć.

Przypomnij sobie treść podrozdziału zatytułowanego „Co się stało z listą plist?”, który prezentowaliśmy nieco wcześniej w tym rozdziale. Skupiliśmy się tam na szczegółowym opisie kroków, które umożliwiły konwersję listy przedstawionej poniżej:

Program eggs.py rozpoczął pracę od tej listy...

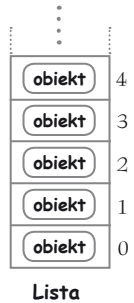
na tę, znacznie krótszą listę:

...i przekształcił ją w tę.

Wszystkie te manipulacje na liście `plist`, wykonane za pomocą metod `pop`, `remove`, `extend` oraz `insert`, zmieniły ją w dość zasadniczy sposób, co nie stanowi zresztą najmniejszego problemu, jako że właśnie to (tj. modyfikacja listy) jest głównym przeznaczeniem tych metod. Jak się jednak sprawy mają z programem *eggs2.py*?

Wycinanie list ma charakter nieniszczący

Metody listy wykorzystywane w programie *eggs.py* do konwersji jednego łańcucha znakowego na drugi miały charakter **niszczący** w tym sensie, że kod, który je zawierał, modyfikował stan listy, na rzecz której działał. Wycinanie ma charakter **nieniszczący**, ponieważ wyodrębnianie obiektów z istniejącej listy nie powoduje zmiany jej stanu, co oznacza, że oryginalne dane pozostają nienaruszone.

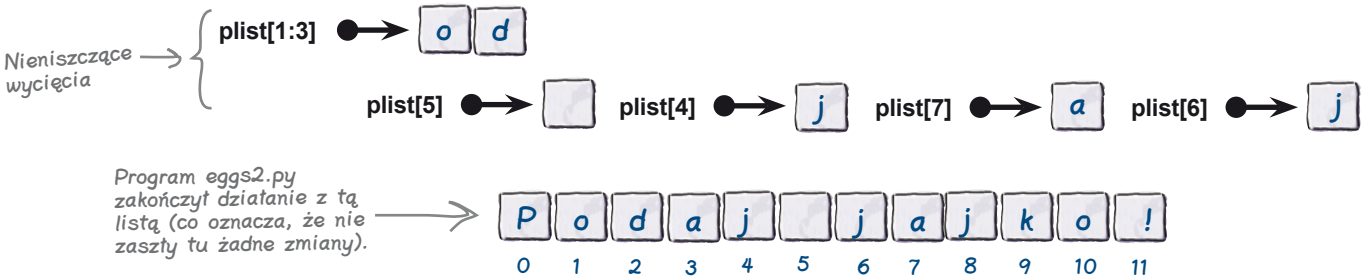


Program eggs2.py rozpoczął działanie z tą listą.

Poniżej pokazaliśmy wycinki wykorzystywane przez kod programu *eggs2.py*. Zauważ, że każda operacja wycinania wyodrębnia dane z listy, ale nie zmienia jej w żaden sposób. Przedstawiliśmy dwa wiersze kodu, które odpowiadają za całą tę brudną robotę, wraz z reprezentacją danych wyodrębnianych przez poszczególne operacje wycinania.

Kod

```
new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
```



A zatem... który z nich jest lepszy?

Stosowanie metod listy w celu manipulowania istniejącą listą i przekształcania jej powoduje dokładnie to, co zostało tu napisane, czyli manipulowanie listą *oraz* jej przekształcanie. Oryginalny stan listy przestaje być dostępny dla Twojego programu. W zależności od tego, co robisz, może to stanowić problem (ale nie musi). Korzystanie z notacji nawiasów kwadratowych zapewnianej przez język Python, ogólnie rzecz biorąc, *nie* zmienia istniejącej listy, chyba że zdecydujesz się przypisać nową wartość do pozycji wskazywanej przez indeks. Stosowanie wycinków również nie modyfikuje listy, co oznacza, że oryginalne dane pozostają nietknięte.

Metody listy zmieniają jej stan, podczas gdy zastosowanie nawiasów kwadratowych i wycinków (zwykle) tego nie robi.

Które z tych dwóch podejść uznasz za „lepsze", zależy głównie od tego, co próbujesz zrobić (nie ma też problemu, jeśli któregoś z nich po prostu nie lubisz). Zawsze istnieje wiele sposobów przeprowadzenia obliczeń, a listy Pythona są wystarczająco elastyczne, aby zapewniać możliwość korzystania z różnych metod interakcji z danymi, które za pomocą tych list przechowujesz.

Dotarliśmy niemal do końca naszej wstępnej prezentacji list. Zostało nam już tylko jedno zagadnienie, które powinieneś poznać na tym etapie: *iterowanie list*.

Pętla for w języku Python „rozumie” listy

Pythonowa pętla `for` „wie” wszystko o listach i — gdy dostarczy jej się *dowolną* listę — „wie” też, gdzie zacząć jej przetwarzanie, ile obiektów zawiera i gdzie znajduje się jej koniec. Nigdy nie musisz podawać pętli `for` którejkolwiek z tych informacji, ponieważ jest ona w stanie sama je uzyskać.

Pomoże nam to zilustrować pewien przykład. Wykonaj go wraz z nami — otwórz nowe okno edycji środowiska IDLE i wpisz kod przedstawiony poniżej. Zapisz ten nowy program jako plik *marvin.py*, po czym naciśnij klawisz *F5*, aby wziąć go w obroty.

Uruchom ten niewielki program...

...aby wygenerować te dane wyjściowe.

marvin.py - C:/Users/lsuma/Desk...

File Edit Format Run Options Window Help

```
paranoid_android = "Marvin"
letters = list(paranoid_android)
for char in letters:
    print('\t', char)
```

Ln: 6 Col: 0

Python 3.6.0 Shell

File Edit Shell Debug Options Window

```
Python 3.6.0 (v3.6.0:41df79263a11,                win32
Type "copyright", "credits" or "li
>>>
============ RESTART: C:/Users/lsuma/Desktop/Python/R02/marvin.py ============
        M
        a
        r
        v
        i
        n
>>>
```

Każdy znak z listy letters został wyświetlony w osobnym wierszu i poprzedzony znakiem tabulacji (za ten ostatni odpowiada w kodzie łańcuch \t).

Ln: 11 Col: 4

Jak działa kod programu marvin.py?

Pierwsze dwa wiersze kodu programu *marvin.py* powinny wydawać Ci się znajome. Odbywa się tu przypisanie łańcucha znakowego do zmiennej (o nazwie `paranoid_android`), a następnie przekształcenie go w listę obiektów znakowych (przypisaną do nowej zmiennej o nazwie `letters`).

Chcemy jednak, żebyś się skoncentrował na kolejnej instrukcji, czyli pętli `for`.

W każdej iteracji pętli `for` wybierany jest kolejny obiekt należący do listy `letters`, a następnie jest on przypisywany do innej zmiennej (o nazwie `char`). We wciętym ciele pętli `for` zmienna ta zawiera bieżącą wartość obiektu, który jest przetwarzany przez tę pętlę. Zauważ, że pętla ta „wie”, gdzie *rozpocząć* iterowanie, kiedy je *zakończyć*, jak również *ile* obiektów znajduje się na liście `letters`. Nie musisz się przejmować żadną z tych informacji, ponieważ jest to zadanie interpretera.

W każdej iteracji zmienna ta odwołuje się do bieżącego obiektu.

To jest lista, po której odbywa się iteracja.

```
for char in letters:
    print('\t', char)
```

Ten blok kodu jest wykonywany w każdej iteracji.

Pętla for w języku Python „rozumie” wycinki

Gdy używasz notacji nawiasów kwadratowych do wybrania wycinka z listy, pętla `for` „robi co trzeba”, iterując tylko po wyciętych obiektach. Pokaże Ci to w akcji nieco zaktualizowana wersja naszego najnowszego programu. Zapisz kopię pliku *marvin.py* pod nazwą *marvin2.py*, po czym zmodyfikuj kod w taki sposób, aby wyglądał tak, jak przedstawiony poniżej.

Interesujący jest sposób, w jaki skorzystaliśmy tu z zapewnianego przez język Python **operatora mnożenia** (*), który używany jest do kontrolowania liczby znaków tabulacji wyświetlanych przed każdym obiektem w drugiej i trzeciej pętli `for`. Zastosowaliśmy tutaj operator *, aby „pomnożyć” liczbę wystąpień tego znaku, tak jak zostało to pokazane poniżej.

marvin2.py - C:/Users/lsuma/Desktop/Python/R02/marvi...

File Edit Format Run Options Window Help

```
paranoid_android = "Marvin, paranoiczny android"
letters = list(paranoid_android)
for char in letters[:6]:
    print('\t', char)
print()
for char in letters[-7:]:
    print('\t'*2, char)
print()
for char in letters[8:19]:
    print('\t'*3, char)
```

Ln: 13 Col: 0

Pierwsza pętla iteruje po wycinku obejmującym sześć pierwszych obiektów na liście.

Druga pętla iteruje po wycinku obejmującym siedem ostatnich obiektów na liście. Zauważ, że zastosowanie konstrukcji *2 spowodowało wstawienie dwóch znaków tabulacji przed każdym z wyświetlanych tu obiektów.

Trzecia (i ostatnia) pętla iteruje po wycinku obejmującym kilka obiektów znajdujących się w środku listy, wybierając znaki, które tworzą słowo „paranoiczny”. Zauważ, że zastosowanie konstrukcji *3 spowodowało wstawienie trzech znaków tabulacji przed każdym z wyświetlanych tu obiektów.

Python 3.6.0 Shell

File Edit Shell D

```
========== RES
        M
        a
        r
        v
        i
        n

                a
                n
                d
                r
                o
                i
                d

                        p
                        a
                        r
                        a
                        n
                        o
                        i
                        c
                        z
                        n
                        y
>>>
```

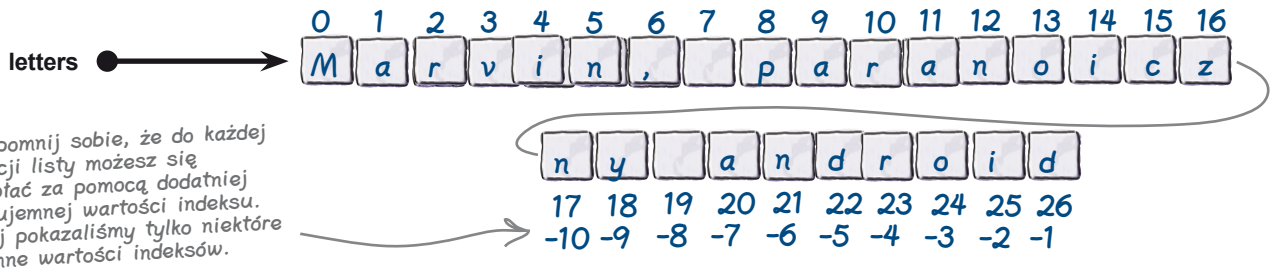
Ln: 63 Col: 4

Wycinki listy Marvina z bliska

Przyjrzymy się dokładniej każdemu z wycinków wykorzystywanych w ostatnim programie, ponieważ zastosowany tu sposób używany jest w wielu programach napisanych w języku Python. Poniżej raz jeszcze został zaprezentowany każdy istotny wiersz kodu programu wraz z graficzną reprezentacją odbywającej się operacji.

Zanim zaczniemy omawiać te trzy wycinki, zauważ, że program rozpoczyna się od przypisania łańcucha znakowego do zmiennej (o nazwie `paranoid_android`) i skonwertowania go na postać listy (o nazwie `letters`).

```
paranoid_android = "Marvin, paranoiczny android"
letters = list(paranoid_android)
```



Teraz przypatrzymy się wszystkim wycinkom, z którymi mieliśmy do czynienia w programie *marvin2.py*, i zobaczymy, jakie dane są przez nie wyodrębniane. Gdy interpreter napotyka specyfikację wycinka, wydobywa odpowiednie obiekty z listy `letters` i przekazuje ich kopie pętli `for`. Oryginalna zawartość tej listy nie zostaje w żaden sposób zmieniona przez te wycinki.

Pierwszy z wycinków wydobywa obiekty od początku listy i kończy to robić na obiekcie znajdującym się na pozycji 6 (nie uwzględniając go jednak).

```
for char in letters[:6]:
    print('\t', char)
```

Drugi wycinek wydobywa obiekty z końca listy `letters`, zaczynając od pozycji –7 i kontynuując do samego końca tej listy.

```
for char in letters[-7:]:
    print('\t'*2, char)
```

I wreszcie na koniec, trzeci wycinek wydobywa obiekty ze środka listy — rozpoczyna od pozycji 8 i obejmuje wszystko aż do pozycji 19, lecz ją samą pomija.

```
for char in letters[8:19]:
    print('\t'*3, char)
```

Listy: odświeżanie tego, co już wiemy

Teraz, gdy przekonałeś się już, jak współpracują ze sobą listy i pętle `for`, przejrzyjmy szybko to, czego dowiedziałeś się z lektury kilku ostatnich stron.

CELNE SPOSTRZEŻENIA

- Listy „rozumieją" notację nawiasów kwadratowych, dzięki czemu można z niej korzystać, aby wybierać pojedyncze obiekty z dowolnej listy.
- Podobnie jak w wielu innych językach programowania, w Pythonie liczenie zaczyna się od zera, dlatego pierwszy obiekt na dowolnej liście znajduje się na pozycji o indeksie 0, drugi — na pozycji o indeksie 1 i tak dalej.
- W przeciwieństwie do wielu innych języków programowania Python umożliwia indeksowanie listy od każdego jej końca. Korzystając z indeksu –1, wskazuje się ostatni element listy, używając –2 — przedostatni i tak dalej.
- Listy zapewniają również obsługę wycinków (lub fragmentów), umożliwiając określenie początku, końca i kroku jako elementów notacji nawiasów kwadratowych.

Widzę, że znajdę mnóstwo zastosowań list w swoich programach napisanych w Pythonie. Ale czy jest coś, w czym listy nie sprawdzają się tak dobrze?

Listy mają mnóstwo zastosowań, ale...

Nie stanowią panaceum na wszystkie potrzeby związane ze strukturami danych. Listy można wykorzystywać w bardzo wielu miejscach. Lista sprawdzi się wręcz doskonale, gdy będziesz miał kolekcję podobnych obiektów, które będziesz chciał przechować w postaci struktury danych.

Jednakże — i być może też dość paradoksalnie — listy mogą okazać się **złym wyborem**, gdy dane, z których będziesz korzystał, będą miały pewną *strukturę*. Problem ten (oraz jego potencjalne rozwiązania) zaczniemy omawiać już na następnej stronie.

Nie istnieją głupie pytania

P: Z pewnością na tym nie kończy się wiedza na temat list?

O: Masz rację, to jeszcze nie wszystko na ich temat. Traktuj materiał przedstawiony w tym rozdziale jako szybkie wprowadzenie do wbudowanych struktur danych Pythona oraz tego, co można za ich pomocą osiągnąć. Na pewno nie udało nam się wyczerpać zagadnienia list i będziemy jeszcze do nich dość często wracać w pozostałej części niniejszej książki.

P: A co z sortowaniem list? Czy nie jest to istotna kwestia?

O: Tak, jest, ale nie zaprzątajmy sobie głowy sprawami tego rodzaju aż do czasu, gdy będzie nam to rzeczywiście potrzebne. Teraz masz już całkiem niezłe pojęcie o podstawach, i to wszystko, co jest Ci niezbędne na tym etapie. I nie martw się na zapas, bo wkrótce zajmiemy się również kwestią sortowania.

Co jest nie tak z listami?

Gdy programiści Pythona odkrywają, że są w sytuacji, w której zachodzi konieczność przechowywania kolekcji podobnych obiektów, zastosowanie listy okazuje się zwykle zupełnie naturalnym wyborem. W końcu w tym rozdziale nie używaliśmy niczego innego.

Przypomnij sobie, jak listy doskonale nadają się do przechowywania kolekcji powiązanych ze sobą liter, co miało miejsce w przypadku listy `vowels`.

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

Jeśli dane tworzy kolekcja liczb, również wtedy listy okazują się świetną opcją.

```
nums = [1, 2, 3, 4, 5]
```

W gruncie rzeczy listy są doskonałym wyborem zawsze wtedy, gdy korzystasz z kolekcji powiązanych ze sobą elementów *dowolnego* rodzaju.

Wyobraź sobie jednak, że musisz przechować dane na temat jakiejś osoby, a wyglądają one mniej więcej tak, jak przedstawiono poniżej.

Z pozoru dane te nie pasują do struktury; tworzą je *etykiety* znajdujące się po lewej stronie oraz *związane z nimi wartości danych* widoczne z prawej. Dlaczego zatem nie umieścić tych danych na liście? Jakby nie patrzeć, dane te są w końcu ze sobą powiązane (przez to, że są związane z pewną osobą), prawda?

Aby przekonać się, dlaczego nie powinniśmy tego robić, przyjrzyjmy się dwóm sposobom zapisywania tych danych na listach (zaczniemy już na następnej stronie). Będziemy tu z Tobą całkowicie szczerzy: *obydwie* nasze próby uwidaczniają problemy, które sprawiają, że zastosowanie w tym przypadku list okazuje się rozwiązaniem dalekim od doskonałego. Ponieważ jednak już sama podróż stanowi połowę radości związanej z dotarciem do celu, to i tak zamierzamy spróbować przebyć tę drogę.

W naszej pierwszej próbie skupimy się na wartościach danych znajdujących się w prawej części serwetki, podczas gdy w drugiej spróbujemy wykorzystać zarówno etykiety z lewej, jak i związane z nimi wartości danych. Pomyśl chwilę, jak za pomocą list mógłbyś poradzić sobie z tego rodzaju danymi strukturalnymi, po czym przejdź na następną stronę, aby przekonać się, jak poszły nasze dwie próby.

Kiedy nie używać list

Mamy swoje przykładowe dane (zapisane na odwrocie serwetki) i zdecydowaliśmy się przechowywać je na liście (ponieważ tylko ją znamy na tym etapie naszej podróży z Pythonem).

Nasza pierwsza próba polega na umieszczeniu wszystkich wartości danych na liście, tak jak zostało to pokazane poniżej.

```
>>> person1 = ['Ford Prefect', 'mężczyzna', 'badacz',
'Betelgeza Siedem']
>>> person1
['Ford Prefect', 'mężczyzna', 'badacz', 'Betelgeza
Siedem']
```

Czy person[1] odnosi się do płci, czy zawodu? Nigdy nie mogę sobie tego przypomnieć!

Skutkuje to otrzymaniem listy obiektów tekstowych, która najwyraźniej działa. Jak widać powyżej, powłoka potwierdziła, że wartości danych znalazły się już na liście o nazwie `person1`.

Mamy tu jednak pewien problem, ponieważ musimy pamiętać, że na pozycji wskazywanej przez pierwszy indeks (czyli indeks 0) znajduje się nazwisko naszej przykładowej osoby, następna jest jej płeć (pod indeksem 1) i tak dalej. Dla niewielkiej liczby elementów danych nie stanowi to wielkiego problemu, wyobraź sobie jednak, że dane rozrastają się tak, że obejmują znacznie więcej wartości (na przykład szczegółowe dane na temat profilowej strony w Twoim nowym, rewelacyjnym serwisie społecznościowym, który zostawi daleko w tyle Facebooka). W takim przypadku korzystanie z wartości indeksów w celu odwoływania się do poszczególnych danych zgromadzonych na liście `person1` jest co najmniej kiepskim pomysłem i lepiej byłoby takiego rozwiązania unikać.

Nazwisko: Ford Prefect
Płeć: mężczyzna
Zawód: badacz
Planeta macierzysta: Betelgeza Siedem

W naszej drugiej próbie postanowiliśmy dodać do listy odpowiednie etykiety, aby każdą wartość danych poprzedzała związaną z nią etykieta. Poznaj listę `person2`.

```
person2 = ['Nazwisko', 'Ford Prefect', 'Płeć', 'mężczyzna',
'Zawód', 'badacz', 'Planeta macierzysta', 'Betelgeza Siedem']
>>> person2
['Nazwisko', 'Ford Prefect', 'Płeć', 'mężczyzna', 'Zawód',
'badacz', 'Planeta macierzysta', 'Betelgeza Siedem']
```

Widać od razu, że to rozwiązanie działa, teraz nie mamy już jednak jednego problemu, lecz dwa. Nie tylko nadal musimy pamiętać, co znajduje się na pozycji wskazywanej przez każdy indeks, lecz również to, że wartości indeksów 0, 2, 4, 6 i tak dalej wskazują etykiety, a wartości indeksów 1, 3, 5, 7 i tak dalej wskazują wartości danych.

Z pewnością musi istnieć jakiś lepszy sposób poradzenia sobie z danymi o strukturze tego rodzaju?

Rzeczywiście, jest. I wymaga zrezygnowania ze stosowania list w przypadku danych strukturalnych tego typu. Trzeba tu użyć czegoś innego, a to coś jest w języku Python określane mianem **słownika** — zajmiemy się nim już w następnym rozdziale.

Jeśli w przypadku danych, które chcesz przechowywać, da się zidentyfikować jakąś strukturę, rozważ zastosowanie innej struktury danych niż lista.

Kod przykładów z rozdziału 2. (1. z 2)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Miliard"
for letter in word:
    if letter in vowels:
        print(letter)
```

Pierwsza wersja programu vowels, wyświetlająca WSZYSTKIE samogłoski z listy, które da się znaleźć w słowie „Miliard" (włącznie z duplikatami).

Program vowels2.py zawiera dodatkowy kod, w którym używana jest lista umożliwiająca wyeliminowanie duplikatów. Program ten wyświetla listę unikatowych samogłosek znalezionych w słowie „Miliard" i należących do listy vowels.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Miliard"
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać
samogłoski: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Trzecia (i ostatnia) wersja programu vowels w tym rozdziale, vowels3.py, wyświetla listę unikatowych samogłosek z listy, które zostały znalezione w słowie wprowadzonym przez użytkownika.

To najlepsze, co można zjeść na śniadanie, więc... „Podaj jajko!". Ten program, o nazwie eggs.py, przyjmuje łańcuch znakowy z powyższą prośbą i za pomocą zestawu metod listy przekształca ten łańcuch w inny, który opisuje, od czego redaktorzy serii „Rusz głową!" zaczynają swój dzień: „od jaj".

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("a")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Kod przykładów z rozdziału 2. (2. z 2)

```
phrase = "Podaj jajko!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

Gdy chodzi o manipulowanie listami, korzystanie z metod nie jest jedyną możliwością. Program eggs2.py wykonuje to samo zadanie, używając notacji nawiasów kwadratowych.

```
paranoid_android = "Marvin"
letters = list(paranoid_android)
for char in letters:
   print('\t', char)
```

Najkrótszy program w tym rozdziale, marvin.py, stanowi przykład tego, jak dobrze listy dogadują się z pętlą for w Pythonie. (Tylko nie mów Marvinowi... Gdyby dowiedział się, że jego program jest najkrótszy w tym rozdziale, mógłby popaść w jeszcze większą paranoję niż teraz).

Program marvin2.py pokazał Ci sposób zastosowania notacji nawiasów kwadratowych Pythona do wybrania trzech wycinków, dzięki którym udało się wyodrębnić i wyświetlić odpowiednie fragmenty listy znaków.

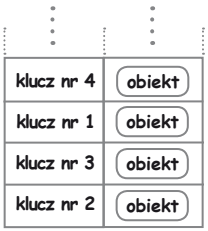
```
paranoid_android = "Marvin, paranoiczny android"
letters = list(paranoid_android)
for char in letters[:6]:
   print('\t', char)
print()
for char in letters[-7:]:
   print('\t'*2, char)
print()
for char in letters[8:19]:
   print('\t'*3, char)
```

Korzystanie z danych strukturalnych

Zapewniana przez język Python struktura danych, jaką jest lista, sprawdza się świetnie, ale nie stanowi remedium na wszelkie potrzeby związane z danymi.

Gdy masz do czynienia z *prawdziwie* strukturalnymi danymi (a używanie listy do ich przechowywania może nie być najlepszą opcją), Python oferuje Ci ratunek w postaci wbudowanego **słownika**. Dostępny od ręki słownik umożliwia przechowywanie wszelkich kolekcji składających się z *par klucz-wartość* i manipulowanie nimi. W tym rozdziale będziemy mu się przyglądać długo i uważnie, a gdzieś po drodze spotkamy też **zbiór** i **krotkę**. Wraz z **listą** (którą zaprezentowaliśmy w poprzednim rozdziale) struktury danych, jakimi są słownik, zbiór i krotka, zapewniają zestaw wbudowanych narzędzi, które sprawiają, że Python i dane tworzą tak potężną kombinację.

Słownik przechowuje pary klucz-wartość



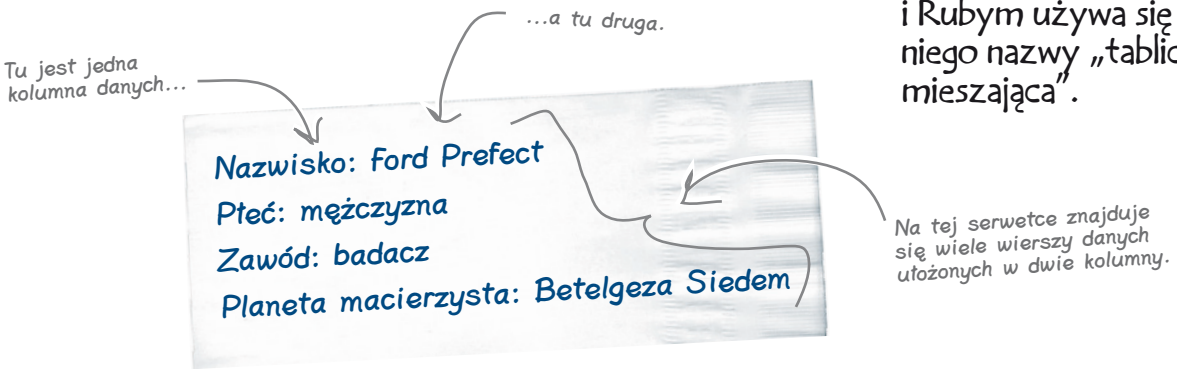
Słownik

W przeciwieństwie do listy, która jest kolekcją związanych ze sobą obiektów, ze **słownika** korzysta się w celu przechowywania kolekcji **par klucz-wartość**, w przypadku której każdy unikatowy *klucz* ma odpowiadającą jej *wartość*. Słownik określa się często w informatyce mianem *tablicy asocjacyjnej*, a w innych językach programowania używa się jeszcze innych, właściwych dla nich nazw (takich jak mapa, tablica mieszająca czy po prostu tabela).

W przypadku słownika w języku Python klucz jest zwykle łańcuchem znakowym, podczas gdy powiązana z nim wartość może być dowolnym obiektem Pythona.

Dane, które pasują do modelu słownika, łatwo jest wskazać: składają się z **dwóch kolumn**, które mogą zawierać **wiele wierszy** danych. Mając to na uwadze, spójrz raz jeszcze na naszą „serwetkę z danymi", z której korzystaliśmy pod koniec poprzedniego rozdziału.

W językach C++ i Java słownik znany jest jako „mapa", a w Perlu i Rubym używa się dla niego nazwy „tablica mieszająca".



Wygląda na to, że dane widniejące na tej serwetce doskonale nadają się do zapisania w słowniku Pythona.

Powróćmy do powłoki, aby przekonać się, jak z tych danych zapisanych na serwetce utworzyć słownik. Kuszące wydaje się wpisanie całego tego słownika w postaci jednego wiersza kodu, jednak tego nie zrobimy. Ponieważ chcemy, aby nasz kod było łatwo czytać, celowo wprowadzimy każdy wiersz danych (czyli każdą parę klucz-wartość) w osobnym wierszu. Przyjrzyj się kodowi przedstawionemu poniżej.

Nazwa słownika (przypomnij sobie, że pod koniec poprzedniego rozdziału mieliśmy do czynienia ze zmiennymi person1 i person2)

Klucz

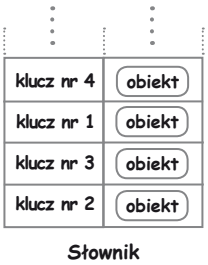
Powiązana wartość danej

```
>>> person3 = { 'Nazwisko': 'Ford Prefect',
                'Płeć': 'mężczyzna',
                'Zawód': 'badacz',
                'Planeta macierzysta': 'Betelgeza Siedem' }
```

Klucz

Wartość

Spraw, aby słowniki łatwo się czytało



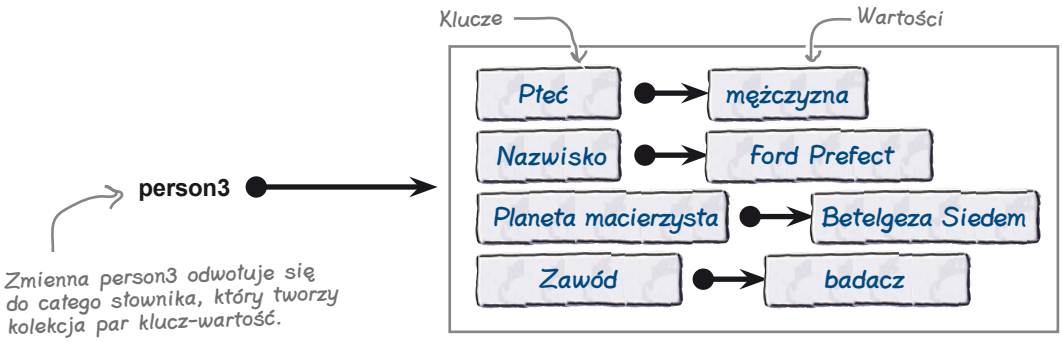
Bardzo kuszącą perspektywą może wydawać się wzięcie czterech wierszy kodu z końca poprzedniej strony i wpisanie ich w powłoce w sposób przedstawiony poniżej.

```
>>> person3 = { 'Nazwisko': 'Ford Prefect',
'Płeć': 'mężczyzna', 'Zawód': 'badacz', 'Planeta
macierzysta': 'Betelgeza Siedem' }
```

Choć interpreter w ogóle nie dba o to, jak wprowadzasz dane, wpisanie ich w taki sposób (a więc jako jeden długi wiersz) utrudnia czytanie kodu i dlatego powinieneś unikać tego za wszelką cenę.

Gdy zaśmiecisz swój kod słownikami, które trudno będzie przeczytać, inni programiści (a wśród nich *Ty* sam za jakieś sześć miesięcy) będą przeklinali na czym świat stoi. Dlatego warto poświęcić nieco czasu na odpowiednie wyrównanie kodu tworzącego Twój słownik, aby *dało się* go łatwo przeczytać.

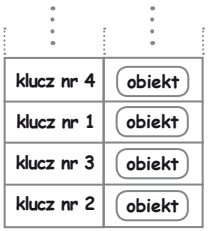
Oto graficzna reprezentacja tego, jak słownik Pythona jest odwzorowywany w pamięci po wykonaniu którejkolwiek z powyższych instrukcji.



Jak widać, jest to nieco bardziej skomplikowana struktura niż przypominająca tablicę lista. Jeśli idea stojąca za pythonowym słownikiem jest dla Ciebie czymś nowym, możesz ją traktować jak **tablicę wyszukiwania** (ang. *lookup table*). Klucz znajdujący się po lewej stronie jest wykorzystywany do *wyszukiwania* wartości po prawej (dokładnie w taki sposób, z jakiego korzystasz, szukając słowa w papierowym słowniku).

Poświęćmy trochę czasu na bliższe poznanie słownika. Zaczniemy od dokładnego wyjaśnienia, jak namierzyć słownik Pythona w swoim kodzie, a następnie przejdziemy do omówienia jego unikatowych cech oraz możliwych zastosowań.

Jak znaleźć słownik w kodzie



Słownik

Przyjrzyj się dokładnie sposobowi, w jaki zdefiniowaliśmy słownik `person3` w powłoce. Na początek zauważ, że *cały* słownik ujęty jest w nawiasy klamrowe. Każdy **klucz** znajduje się tu w pojedynczych cudzysłowach, podobnie zresztą jak każda **wartość**. Zarówno klucze, jak i wartości są w naszym przykładzie łańcuchami znakowymi (choć bynajmniej nie muszą). Każdy klucz jest oddzielony od powiązanej z nim wartości za pomocą znaku **dwukropka** (`:`), a każda para klucz-wartość (znana również jako „wiersz") oddzielona jest od następnej **przecinkiem**.

Każdy słownik zaczyna się od otwierającego nawiasu klamrowego.

Wszystkie wartości przechowywane w tym słowniku są łańcuchami znakowymi, dlatego zostały ujęte w pojedyncze cudzysłowy.

```
{ 'Nazwisko': 'Ford Prefect',
  'Płeć': 'mężczyzna',
  'Zawód': 'badacz',
  'Planeta macierzysta': 'Betelgeza Siedem' }
```

Każda para klucz-wartość jest oddzielona od następnej znakiem przecinka.

Każdy klucz jest ujęty w cudzysłów.

Dwukropek łączy każdy klucz z jego wartością.

Każdy słownik kończy się znakiem zamykającego nawiasu klamrowego.

Jak już wspominaliśmy wcześniej, dane zapisane na serwetce świetnie pasują do słownika Pythona. W gruncie rzeczy każde dane o podobnej strukturze — tj. złożone z wielu wierszy podzielonych na dwie kolumny — będą najlepszymi kandydatami do umieszczenia w słowniku, jakich tylko możesz znaleźć. Doskonale, ale to ma też swoją cenę. Wróćmy do okna powłoki, aby przekonać się jaką.

Poproś powłokę o wyświetlenie zawartości słownika...

```
>>> person3
{'Płeć': 'mężczyzna', 'Nazwisko': 'Ford Prefect', 'Planeta
macierzysta': 'Betelgeza Siedem', 'Zawód': 'badacz'}
```

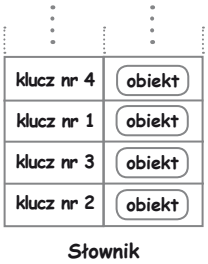
...i oto on. Wszystkie pary klucz-wartość zostały pokazane.

Co się stało z kolejnością, w jakiej zostały wprowadzone dane?

Przyjrzyj się z bliska słownikowi wyświetlonemu przez interpreter. Czy zauważyłeś, że kolejność danych jest tu zupełnie inna niż ta, w której je pierwotnie wprowadziłeś? Gdy tworzyłeś słownik, wpisywałeś wiersze w następującej kolejności: nazwisko, płeć, zawód, planeta macierzysta. Jednak powłoka pokazała je w innej, a mianowicie: płeć, nazwisko, planeta macierzysta, zawód. Porządek się zmienił.

Co się tu dzieje? Dlaczego kolejność uległa zmianie?

Kolejność wprowadzania NIE jest zachowywana



W przeciwieństwie do list, które zachowują porządek elementów zastosowany przy ich wprowadzaniu, słowniki Pythona tego **nie** robią. Oznacza to, że nie możesz założyć, że wiersze w jakimkolwiek słowniku są ułożone w jakiejś określonej kolejności. Co by się miało nie dziać, zawsze są one **nieuporządkowane**.

Spójrz jeszcze raz na słownik `person3` i porównaj porządek wprowadzania elementów do tego porządku, w jakim interpreter wyświetlił dane w powłoce.

```
>>> person3 = { 'Nazwisko': 'Ford Prefect',
                'Płeć': 'mężczyzna',
                'Zawód': 'badacz',
                'Planeta macierzysta': 'Betelgeza Siedem' }
>>> person3
{'Płeć': 'mężczyzna', 'Nazwisko': 'Ford Prefect', 'Planeta
macierzysta': 'Betelgeza Siedem', 'Zawód': 'badacz'}
```

Wprowadzasz swoje dane do słownika w określonej kolejności...

...ale interpreter wyświetla je w zupełnie innej.

Jeśli drapiesz się teraz w głowę i zastanawiasz, dlaczego miałbyś powierzać swoje drogocenne dane tak nieuporządkowanej strukturze danych, nie martw się, ponieważ kolejność elementów rzadko robi tu jakąś większą różnicę. Gdy decydujesz się zapisać dane w słowniku, nie ma dla nich znaczenia kolejność, w jakiej są ułożone, zasadnicze znaczenie mają natomiast zastosowane klucze. Pamiętaj, że to klucz jest wykorzystywany do wyszukiwania wartości.

Słowniki „rozumieją" nawiasy kwadratowe

Podobnie jak listy, słowniki obsługują notację nawiasów kwadratowych. W przeciwieństwie jednak do list, w przypadku których, aby odwołać się do poszczególnych elementów, używa się indeksów o wartościach liczbowych, w słownikach w celu uzyskania dostępu do powiązanych wartości korzysta się z kluczy. Sprawdźmy działanie tego mechanizmu, korzystając z powłoki.

```
>>> person3['Planeta macierzysta']
'Betelgeza Siedem'

>>> person3['Nazwisko']
'Ford Prefect'
```

Podaj klucz w nawiasach kwadratowych.

Na ekranie pojawi się wartość danej powiązanej z tym kluczem.

Korzystaj z kluczy, aby uzyskać dostęp do danych przechowywanych w słowniku.

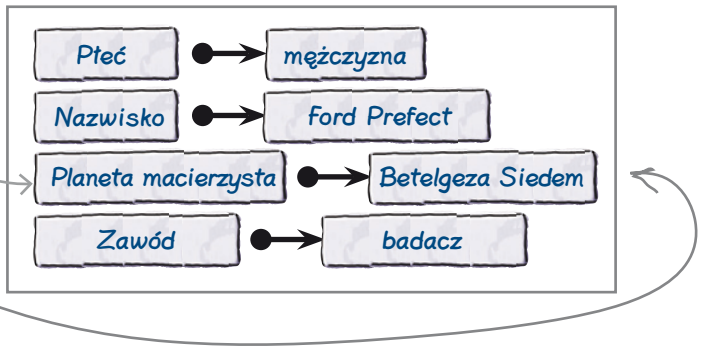
Gdy uświadomisz sobie, że w ten sposób możesz uzyskać dostęp do swoich danych, jasne stanie się, że kolejność, w jakiej interpreter je zapisuje, nie ma najmniejszego znaczenia.

Wyszukiwanie wartości przy użyciu nawiasów kwadratowych

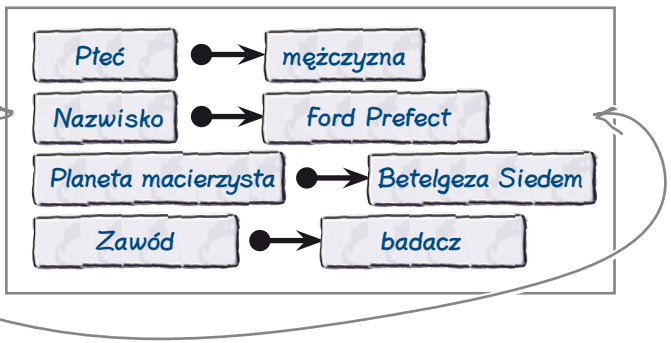
Sposób korzystania z nawiasów kwadratowych w przypadku słowników jest dokładnie taki sam jak w przypadku list. Zamiast jednak odwoływać się do danych znajdujących się na określonej pozycji za pomocą wartości indeksu, w słowniku Pythona do danych odwołuje się za pomocą powiązanego z nią klucza.

Jak mieliśmy okazję zobaczyć na końcu poprzedniej strony, gdy wewnątrz nawiasów kwadratowych znajdujących się obok nazwy słownika umieści się klucz, interpreter zwróci powiązaną z tym kluczem wartość. Aby utrwalić tę ideę w Twojej głowie, raz jeszcze przyjrzyjmy się przedstawionym wcześniej przykładom.

```
>>> person3['Planeta macierzysta']
'Betelgeza Siedem'
```




```
>>> person3['Nazwisko']
'Ford Prefect'
```



Słowniki przeszukuje się szybko!

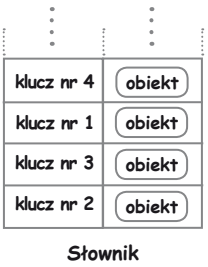
Cechą, która sprawia, że słowniki Pythona są tak przydatne, jest właśnie możliwość wydobywania dowolnej wartości ze słownika za pomocą powiązanego z nią klucza. Dzieje się tak, ponieważ istnieje bardzo wiele sytuacji, w których możliwość ta znajduje zastosowanie. Przykładem może tu być wyszukiwanie szczegółowych danych o użytkowniku w jego profilu, czyli dokładnie ta operacja, którą przeprowadziliśmy powyżej, korzystając ze słownika `person3`.

Kolejność przechowywania danych w słowniku nie ma najmniejszego znaczenia. Naprawdę liczy się tylko to, czy interpreter może *szybko* odwołać się do wartości powiązanej z kluczem (niezależnie od wielkości słownika). Dobra dla nas nowina głosi, że dzięki zastosowaniu wysoce zoptymalizowanego *algorytmu mieszającego* tak właśnie jest. Podobnie jak większością wewnętrznych mechanizmów Pythona, również tym mechanizmem nie musisz się szczególnie przejmować. Troskę o szczegóły pozostaw interpreterowi, a samemu zaś jedynie korzystaj z możliwości oferowanych przez słownik.

Porady maniaka

Słownik Pythona został zaimplementowany jako tablica mieszająca o zmiennym rozmiarze, którą silnie zoptymalizowano pod kątem występowania wielu szczególnych przypadków. Dzięki temu słowniki wyszukują dane bardzo szybko i sprawnie.

Korzystanie ze słowników w czasie wykonania programu



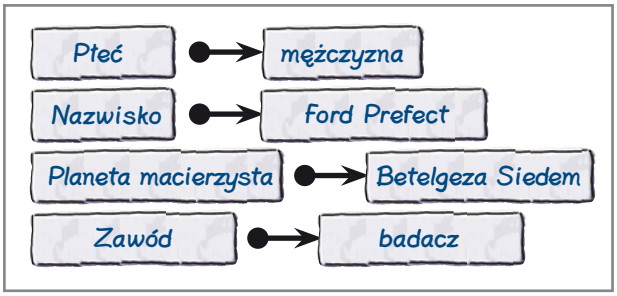
Wiedza na temat tego, jak notacja nawiasów kwadratowych działa w odniesieniu do słowników, ma kluczowe znaczenie dla zrozumienia sposobu ich powiększania w czasie wykonania programu. Jeśli dysponujesz istniejącym słownikiem, możesz dodać do niego nową parę klucz-wartość, przypisując obiekt do nowego klucza podanego w nawiasach kwadratowych.

Poniżej wyświetlamy na przykład bieżący stan słownika `person3`, a następnie dodajemy nową parę klucz-wartość, która łączy wartość 33 z kluczem o nazwie `Wiek`. Potem wyświetlamy słownik `person3` po raz drugi, aby potwierdzić, że nowy wiersz danych został do niego poprawnie dodany.

Przed dodaniem nowego wiersza

```
>>> person3
{'Nazwisko': 'Ford Prefect', 'Płeć': 'mężczyzna', 'Planeta
macierzysta': 'Betelgeza Siedem', 'Zawód': 'badacz'}
```

Przed



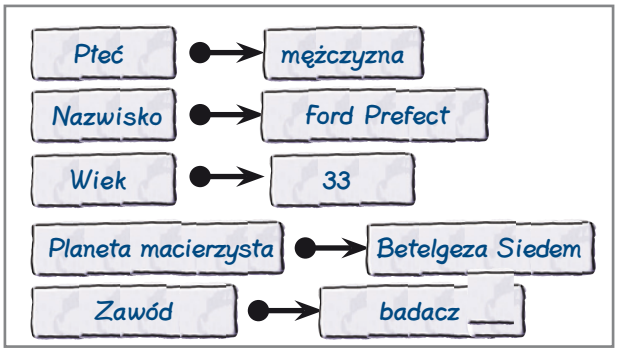

```
>>> person3['Wiek'] = 33
```

Przypisz obiekt (w tym przypadku liczbę) do nowego klucza, aby dodać wiersz danych do słownika.

```
>>> person3
{'Nazwisko': 'Ford Prefect', 'Płeć': 'mężczyzna',
'Wiek': 33, 'Zawód': 'badacz', 'Planeta macierzysta':
'Betelgeza Siedem'}
```

Po dodaniu nowego wiersza

Oto nowy wiersz danych: liczba 33 jest powiązana z kluczem 'Wiek'.



Po

Przypomnienie: wyświetlanie znalezionych samogłosek (przy użyciu list)

Jak zostało to pokazane na poprzedniej stronie, ten sposób powiększania słownika może zostać wykorzystany w wielu różnych sytuacjach. Jednym z bardzo typowych zastosowań jest operacja *zliczania częstotliwości*, polegająca na przetworzeniu pewnych danych w celu otrzymania liczby wystąpień tego, co udało się znaleźć. Przed przystąpieniem do prezentacji przeprowadzania tego działania za pomocą słownika wróćmy na chwilę do przedstawionego w poprzednim rozdziale przykładu, w którym liczyliśmy samogłoski.

Przypomnij sobie, że program *vowels3.py* umożliwiał otrzymanie listy unikatowych samogłosek, które udało się znaleźć w pewnym słowie. Wyobraź sobie teraz, że poproszono Cię o takie rozbudowanie tego programu, aby informował on również o tym, ile razy w sprawdzanym słowie występuje każda z samogłosek.

Poniżej widoczny jest znany Ci z rozdziału 2. kod, który dla danego słowa wyświetla listę unikatowych samogłosek, jakie udało się znaleźć w tym słowie.

To program vowels3.py, który wyświetla unikatowe samogłoski znalezione w podanym słowie.

vowels3.py - C:/Users/lsuma/Desktop/Python/R02/vowels3.py (3.6.0)

File Edit Format Run Options Window Help

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

Przypomnij sobie, że uruchomiliśmy ten kod kilka razy w środowisku IDLE.

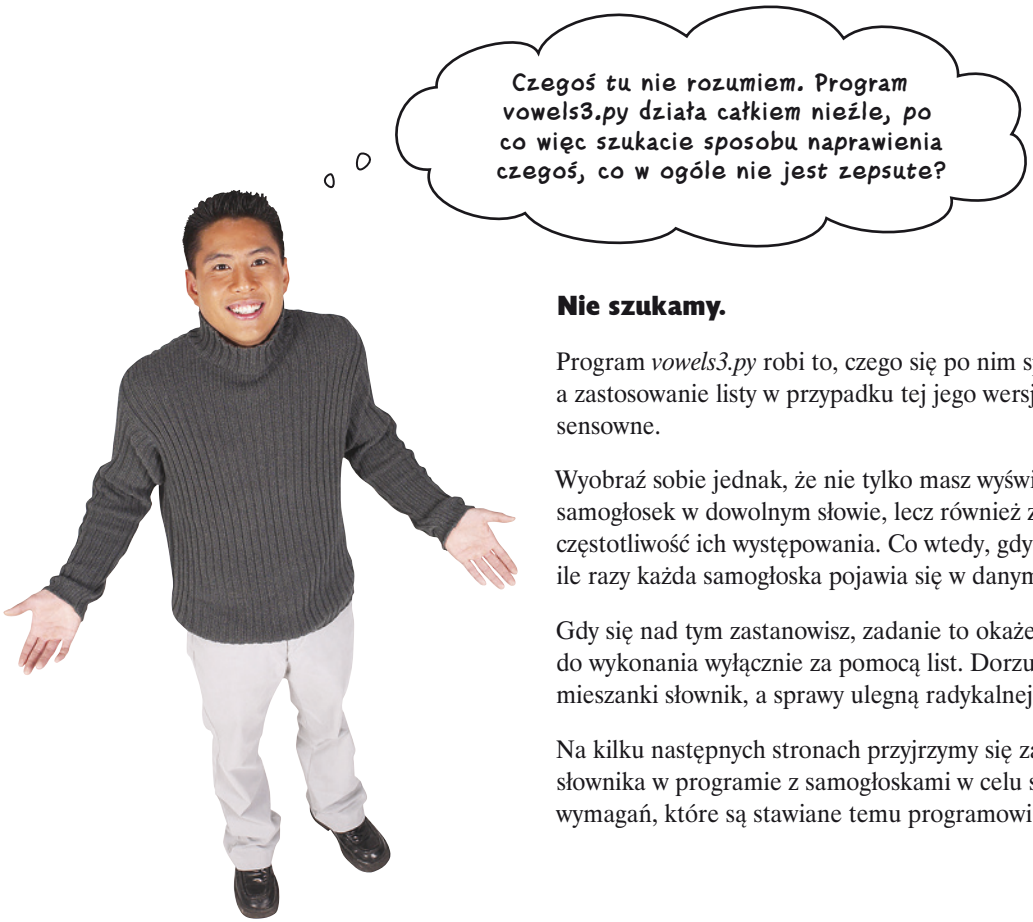
Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Miliard
i
a
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Mienie
i
e
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Mama
a
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels3.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: Rym
>>>
```

Ln: 20 Col: 4

Jak może nam tu pomóc słownik?



Nie szukamy.

Program *vowels3.py* robi to, czego się po nim spodziewamy, a zastosowanie listy w przypadku tej jego wersji jest całkowicie sensowne.

Wyobraź sobie jednak, że nie tylko masz wyświetlić listę samogłosek w dowolnym słowie, lecz również zaraportować częstotliwość ich występowania. Co wtedy, gdy musisz wiedzieć, ile razy każda samogłoska pojawia się w danym słowie?

Gdy się nad tym zastanowisz, zadanie to okaże się dość trudne do wykonania wyłącznie za pomocą list. Dorzuć jednak do tej mieszanki słownik, a sprawy ulegną radykalnej zmianie.

Na kilku następnych stronach przyjrzymy się zastosowaniu słownika w programie z samogłoskami w celu spełnienia nowych wymagań, które są stawiane temu programowi.

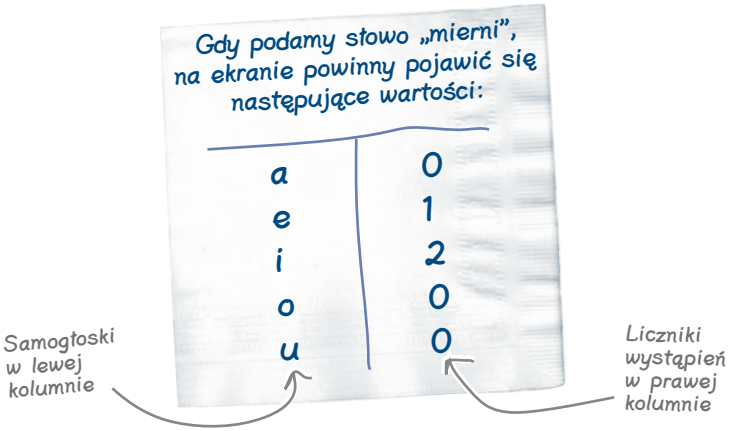
Nie istnieją głupie pytania

P: Czy tylko mnie się wydaje, że wyraz „słownik" to dość dziwna nazwa dla czegoś, co w gruncie rzeczy jest po prostu tabelą?

O: Nie, nie tylko Tobie się tak wydaje. Wyraz „słownik" to pojęcie wykorzystywane w dokumentacji Pythona. Tak naprawdę większość anglojęzycznych programistów używających tego języka zamiast pełnego angielskiego wyrazu „dictionary" posługuje się na co dzień skrótem „dict". W swojej najprostszej formie słownik jest tabelą, która ma dokładnie dwie kolumny i dowolnie wiele wierszy.

Wybór struktury danych do przechowywania częstotliwości

Naszym celem jest taka modyfikacja programu *vowels3.py*, aby informował nas o liczbie wystąpień każdej z samogłosek znalezionych w podanym słowie, a więc ich częstotliwości, tak? Naszkicujmy dane wyjściowe, jakich możemy się spodziewać na ekranie po uruchomieniu tego programu.



Pokazane powyżej dane wyjściowe doskonale odpowiadają temu, w jaki sposób interpreter traktuje słownik. Zamiast więc korzystać z listy do przechowywania znalezionych samogłosek (jak ma to miejsce w przypadku programu *vowels3.py*), zastosujmy właśnie słownik. W stosunku do naszej kolekcji nadal możemy posługiwać się nazwą `found`, ale tym razem musimy ją zainicjalizować za pomocą pustego słownika zamiast za pomocą pustej listy.

Jak zawsze, korzystając z powłoki, poeksperymentujmy trochę i sprawdźmy, czego potrzebujemy, zanim przystąpimy do wprowadzania zmian w kodzie programu *vowels3.py*. Aby utworzyć pusty słownik, przypiszmy `{}` do zmiennej, jak zostało to pokazane poniżej.

```
>>> found = {}
>>> found
{}
```

Samotne nawiasy klamrowe oznaczają, że słownik jest na początku pusty.

Odnotujmy teraz fakt, że nie znaleźliśmy jeszcze żadnej samogłoski. Zróbmy to, tworząc wiersz dla każdej samogłoski i inicjalizując powiązaną z nim wartość za pomocą zera. Każda z naszych samogłosek zostaje użyta w charakterze klucza, co pokazano poniżej.

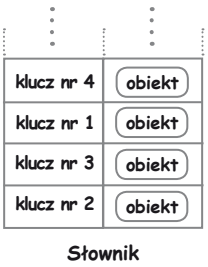
```
>>> found['a'] = 0
>>> found['e'] = 0
>>> found['i'] = 0
>>> found['o'] = 0
>>> found['u'] = 0
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
```

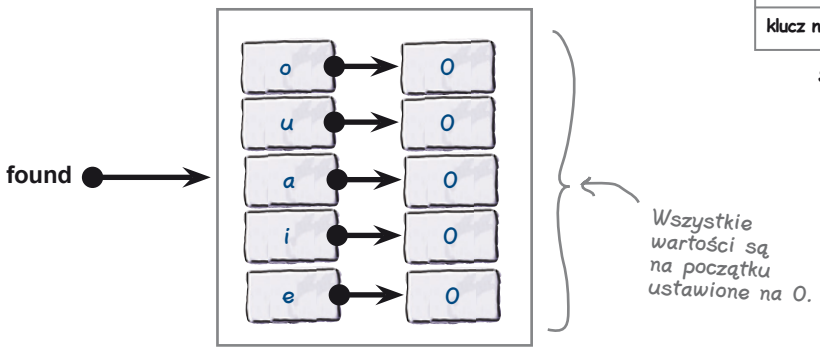
Zainicjalizowaliśmy liczniki wszystkich samogłosek wartością 0. Zwróć uwagę na to, że kolejność wprowadzania poszczególnych wierszy nie została zachowana przez interpreter (ale nie ma to tutaj dla nas najmniejszego znaczenia).

Wszystko, co teraz trzeba zrobić, to odszukać samogłoskę w podanym słowie, a następnie zaktualizować odpowiedni licznik.

Aktualizacja licznika częstotliwości

Zanim zajmiemy się kodem odpowiedzialnym za aktualizację poszczególnych liczników, zastanów się, jak interpreter „widzi” słownik `found` w pamięci po wykonaniu kodu inicjalizującego tę strukturę.





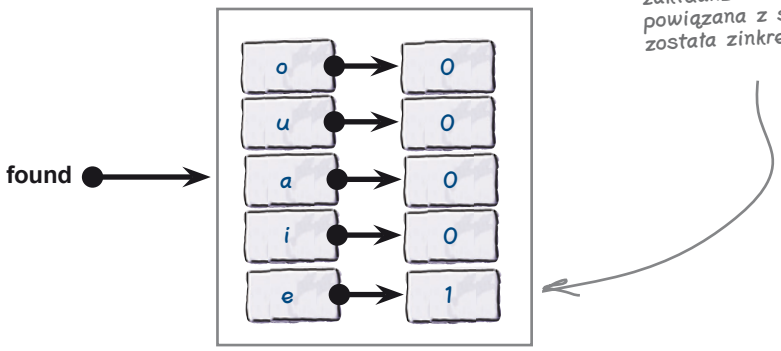
Mając liczniki zainicjalizowane zerami, nietrudno jest odpowiednio zinkrementować dowolną wartość. Poniżej znajdziesz na przykład kod, dzięki któremu można zwiększyć licznik wystąpień samogłoski e.

```
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
>>> found['e'] = found['e'] + 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 1}
```

Wszystko jest zerem.

Inkrementacja licznika litery „e”

Słownik został zaktualizowany. Wartość powiązana z samogłoską „e” została zinkrementowana.



Kod podobny do wyróżnionego powyżej z pewnością działa, ale powtarzanie konstrukcji `found['e']` z obydwu stron operatora przypisania szybko wychodzi z mody w obecnych czasach. Przyjrzyj się zatem, jak można skrócić zapis tej operacji (na następnej stronie).

Aktualizacja licznika częstotliwości, wersja 2.0

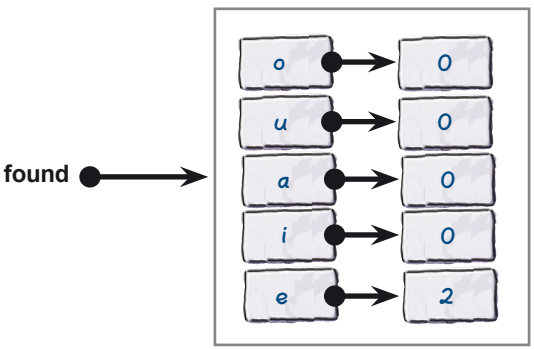
Konieczność umieszczania konstrukcji `found['e']` z obydwu stron operatora przypisania (=) szybko staje się męcząca, dlatego Python umożliwia stosowanie znanego wielu programistom operatora +=, który wykonuje dokładnie to samo działanie, ale znacznie skraca zapis operacji.

```
>>> found['e'] += 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 2}
```

Ponownie inkrementujemy licznik odpowiadający literze „e".

Słownik zostaje ponownie zaktualizowany.

Do tej chwili zinkrementowaliśmy wartość powiązaną z kluczem e dwukrotnie, zatem słownik wygląda teraz „w oczach" interpretera tak, jak widać to poniżej.



Dzięki użyciu operatora += wartość powiązana z kluczem 'e' została zinkrementowana ponownie.

Nie istnieją głupie pytania

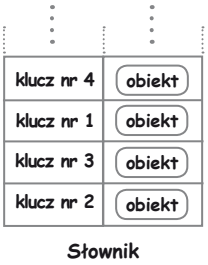
P: Czy Python obsługuje operator ++?

O: Nie... i jest to duże niedopatrzenie. Jeśli jesteś fanem operatora inkrementacji ++ znanego z innych języków programowania, będziesz po prostu musiał przywyknąć do używania zamiast niego operatora +=. Tak samo będzie z operatorem dekrementacji `--`, ponieważ on również nie występuje w Pythonie. Zamiast z niego trzeba korzystać z operatora `-=`.

P: Czy jest gdzieś dostępna jakaś wygodna w użyciu lista operatorów?

O: Tak. Znajdziesz ją pod adresem *https://docs.python.org/3/reference/lexical_analysis.html#operators*. Następnie odwiedź stronę *https://docs.python.org/3/library/stdtypes.html*, aby zapoznać się ze szczegółowym opisem sposobów użycia operatorów w stosunku do typów wbudowanych języka Python.

Iterowanie po słowniku



Do tej pory udało nam się pokazać Ci metodę inicjalizowania słownika za pomocą zer, jak również jego aktualizacji poprzez inkrementowanie wartości powiązanej z kluczem. Jesteśmy już niemal gotowi do modyfikacji programu *vowels3.py* w taki sposób, aby zliczał on wystąpienia poszczególnych samogłosek znalezionych w podanym słowie. Zanim to jednak zrobimy, sprawdźmy, jak odbywa się iterowanie po słowniku, ponieważ gdy będzie on już wypełniony danymi, będzie nam potrzebny jakiś sposób wyświetlania wartości wszystkich liczników na ekranie.

Litościwie wybaczymy Ci pomysł, że wystarczy tu już tylko zaprząc pętlę `for` do iteracji słownika, choć realizacja takiej koncepcji może doprowadzić do uzyskania dość niespodziewanych efektów, co pokazano poniżej.

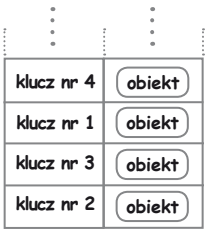
Iterujemy po słowniku w typowy sposób, używając pętli for. Korzystamy tu ze zmiennej o nazwie „kv", która stanowi skrót od angielskich słów „key-value" oznaczających parę klucz-wartość (ale równie dobrze moglibyśmy zastosować dowolną inną nazwę w przypadku tej zmiennej).

```
>>> for kv in found:
    print(kv)

o
i
a
u
e
```

Rozwiązanie zadziałało, ale nie to spodziewaliśmy się ujrzeć na ekranie. Gdzie podziały się wszystkie liczniki wystąpień? Tutaj widać jedynie same klucze...

Przejdź do następnej strony, aby dowiedzieć się, co się stało z wartościami.

Iterowanie po kluczach i wartościach



Słownik

Gdy iterowałeś po słowniku za pomocą pętli `for`, interpreter przetwarzał wyłącznie klucze tego słownika.

Aby odwoływać się również do powiązanych z nimi danych, każdy z kluczy musisz umieścić w nawiasach kwadratowych i używać wraz z nazwą słownika, co umożliwi Ci dostęp do odpowiednich wartości.

Dokładnie to robi pętla przedstawiona poniżej, wyświetlając nie tylko same klucze, lecz również powiązane z nimi wartości danych. Zmodyfikowaliśmy tu zestaw w taki sposób, aby za pomocą każdego klucza występującego w pętli `for` uzyskać dostęp do każdej wartości.

Jako że pętla `for` iteruje po każdej parze klucz-wartość należącej do słownika, w każdym przejściu klucz bieżącego wiersza jest najpierw przypisywany do zmiennej `k`, a dostęp do powiązanej z nim wartości uzyskuje się następnie dzięki konstrukcji `found[k]`. Do wywołania funkcji `print` dodaliśmy też dwa łańcuchy znakowe, aby nieco ułatwić Ci odczytanie danych wyjściowych z ekranu.

Używamy zmiennej k do reprezentowania klucza, a konstrukcji found[k] w celu uzyskania dostępu do wartości.

```
>>> for k in found:
    print(k, 'znaleziono', found[k], 'raz(y).')

o znaleziono 0 raz(y).
i znaleziono 0 raz(y).
a znaleziono 0 raz(y).
u znaleziono 0 raz(y).
e znaleziono 2 raz(y).
```

To wygląda już o niebo lepiej. Zarówno klucze, jak i wartości zostały przetworzone przez pętlę i wyświetlone na ekranie.

Jeśli podczas czytania wykonujesz poszczególne operacje w swojej powłoce, a Twoje dane wyjściowe są ułożone na ekranie w innej kolejności niż nasze, nie martw się. Gdy wykonujesz poszczególne operacje na słowniku, interpreter wewnętrznie układa jego dane w sposób losowy i nie gwarantuje tu zastosowania jakiegokolwiek określonego porządku. Kolejność Twoich danych prawdopodobnie będzie się różniła od tej, którą zaobserwowaliśmy my, nie powinieneś jednak się tym niepokoić. Naszą główną troską jest to, aby dane były bezpiecznie przechowywane w słowniku, i tak właśnie jest.

Przedstawiona powyżej pętla w *oczywisty* sposób działa. Jest jeszcze jednak kilka rzeczy, o które powinniśmy tu zadbać.

Po pierwsze, byłoby miło, gdyby dane wyjściowe były uporządkowane alfabetycznie (tj. w kolejności: `a`, `e`, `i`, `o`, `u`), a nie losowo, jak teraz, prawda?

Po drugie, mimo że wyraźnie widać, iż pętla ta działa, zastosowany tu sposób iterowania po słowniku nie jest preferowany, co oznacza, że większość programistów Pythona zrobiłaby to nieco inaczej.

Omówmy obydwa te punkty nieco dokładniej (najpierw jednak zróbmy mały przegląd przedstawionego do tej pory materiału).

Słowniki: co już wiemy

Poniżej znajdziesz podsumowanie wiadomości, które udało nam się do tej pory zdobyć na temat struktury danych, jaką jest słownik.

CELNE SPOSTRZEŻENIA

- Traktuj słownik jak kolekcję wierszy, z których każdy ma dokładnie dwie kolumny. Pierwsza z nich przechowuje **klucz**, podczas gdy druga zawiera **wartość**.
- Każdy wiersz określany jest mianem **pary klucz-wartość**, a słownik może się powiększać w taki sposób, aby zawierać dowolną liczbę takich par. Podobnie jak listy, słowniki mogą się rozrastać lub kurczyć w zależności od potrzeb.
- Słownik łatwo jest znaleźć w kodzie: jest umieszczony w nawiasach klamrowych, każda znajdująca się w nim para klucz-wartość jest oddzielona przecinkiem od kolejnej, a każdy klucz jest oddzielony od odpowiadającej mu wartości za pomocą dwukropka.
- Słownik *nie* zachowuje porządku, w jakim wstawiane są wiersze. Kolejność umieszczania wierszy nie wpływa w żaden sposób na to, jak są one zapisywane w pamięci.
- Do danych przechowywanych przez słownik można się odwołać, korzystając z **notacji nawiasów kwadratowych**. Aby uzyskać dostęp do określonej wartości, należy umieścić w takich nawiasach odpowiadający jej klucz.
- Zapewniana przez Pythona pętla `for` może być wykorzystywana do iterowania po słowniku. Przy każdej iteracji klucz zostaje przypisany do zmiennej pętli, która jest używana do odwoływania się do wartości danej..

Określanie porządku słownika na wyjściu

Chcemy mieć możliwość wyświetlenia danych przetwarzanych w pętli `for` w wybranej przez nas kolejności (`a`, `e`, `i`, `o`, `u`) zamiast w kolejności losowej, jak miało to miejsce do tej pory. Język Python sprawia, że zadanie to jest wręcz trywialne, a to dzięki istnieniu wbudowanej funkcji `sorted`. Wystarczy, że tworząc swoją pętlę `for`, jako argument wywołania przekażesz tej funkcji słownik `found`, a dane wyjściowe zostaną ułożone alfabetycznie, tak jak zostało to przedstawione poniżej.

```
>>> for k in sorted(found):
    print(k, 'znaleziono', found[k], 'raz(y).')

a znaleziono 0 raz(y).
e znaleziono 2 raz(y).
i znaleziono 0 raz(y).
o znaleziono 0 raz(y).
u znaleziono 0 raz(y).
```

To bardzo niewielka modyfikacja kodu pętli, ale... daje całkiem sporo. Spójrz tylko: dane wyjściowe są posortowane w kolejności alfabetycznej.

To pierwsza rzecz, którą mieliśmy zamiar się zająć. Teraz dowiemy się, jakie podejście do tworzenia kodu *preferuje* większość programistów Pythona ponad to, które zostało pokazane powyżej na tej stronie (choć i to ostatnie jest często stosowane w praktyce, dlatego zdecydowanie powinieneś wiedzieć również o nim).

Iterowanie po słowniku przy użyciu metody items

Przekonaliśmy się, że korzystając z kodu widocznego poniżej, można iterować po wierszach danych znajdujących się w słowniku.

```
>>> for k in sorted(found):
    print(k, 'znaleziono', found[k], 'raz(y).')

a znaleziono 0 raz(y).
e znaleziono 2 raz(y).
i znaleziono 0 raz(y).
o znaleziono 0 raz(y).
u znaleziono 0 raz(y).
```

Podobnie jak listy, słowniki oferują sporo wbudowanych metod, a jedną z nich jest metoda `items`, która zwraca listę par klucz-wartość. Użycie tej metody w połączeniu z pętlą `for` stanowi zwykle preferowany sposób iterowania po słowniku, ponieważ daje Ci on dostęp do klucza *oraz* wartości jako zmiennych pętli, z których możesz następnie korzystać w swoim zestawie. Dzięki temu zestaw ten ma nieco prostszą konstrukcję, a to z kolei powoduje, że łatwiej się go czyta.

Poniżej znajdziesz nową wersję powyższego kodu pętli; w wersji tej zastosowano metodę `items`. Zauważ, że mamy tutaj do czynienia z *dwoma* zmiennymi pętli (`k` oraz `v`), a także że nadal korzystamy z funkcji `sorted`, aby kontrolować kolejność, w jakiej wyświetlane są dane.

```
>>> for k, v in sorted(found.items()):
    print(k, 'znaleziono', v, 'raz(y).')

a znaleziono 0 raz(y).
e znaleziono 2 raz(y).
i znaleziono 0 raz(y).
o znaleziono 0 raz(y).
u znaleziono 0 raz(y).
```

Wywołujemy metodę items na rzecz słownika found.

Metoda items przekazuje z powrotem dwie zmienne pętli.

...ale ten kod o wiele łatwiej się czyta.

Te same dane wyjściowe co wcześniej...

Nie istnieją głupie pytania

P: Dlaczego w przypadku drugiej pętli ponownie wywołujemy metodę `sorted`? Przecież pierwsza pętla ułożyła słownik w kolejności, o którą nam chodziło, a więc z pewnością nie musimy sortować jej po raz kolejny, prawda?

O: Nie, nie całkiem. Wbudowana funkcja `sorted` nie zmienia porządku danych, które jej dostarczasz, lecz jedynie zwraca **uporządkowaną kopię** tych danych. W przypadku słownika `found` jest to uporządkowana kopia wszystkich par klucz-wartość, w której kolejność elementów ustalona jest na podstawie klucza (alfabetycznie, od A do Z). Pierwotny porządek słownika pozostaje nienaruszony, co oznacza, że za każdym razem, gdy trzeba iterować po parach klucz-wartość w pewnej szczególnej kolejności, należy wywołać funkcję `sorted`, ponieważ w samym słowniku nadal panuje porządek losowy.

Magnesiki do zliczania wystąpień

Teraz, gdy podsumowaliśmy już wyniki naszych eksperymentów w powłoce, przyszedł czas, aby wprowadzić zmiany w programie *vowels3.py*. Poniżej znajdziesz wszystkie fragmenty kodu, które naszym zdaniem mogą Ci być potrzebne. Twoim zadaniem jest ułożenie magnesików w taki sposób, aby powstał z nich działający kod, który zlicza wystąpienia wskazanych samogłosek w podanym słowie.

Zdecyduj, który magnesik z kodem ma trafić w określone miejsce oznaczone kropkowaną linią, aby powstał program vowels4.py.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
.......................................
.......................................
.......................................
.......................................
.......................................
.......................................
.......................................

for letter in word:
    if letter in vowels:
        ...................... .....................

for ..................... in sorted(.......................................):
    print(.............., 'znaleziono', .............., 'raz(y).')
```

Gdzie powinny się znaleźć poszczególne fragmenty kodu? Bądź ostrożny, ponieważ nie wszystkie te magnesiki są tutaj potrzebne.

- `found = {}`
- `found[letter]`
- `k, v`
- `k`
- `found = []`
- `found['a'] = 0`
  `found['e'] = 0`
  `found['i'] = 0`
  `found['o'] = 0`
  `found['u'] = 0`
- `value`
- `key`
- `found`
- `v`
- `found.items()`
- `+= 1`

Gdy umieścisz już odpowiednie magnesiki w miejscach, które według Ciebie powinny zająć, otwórz kod programu *vowels3.py* w oknie edycji środowiska IDLE, zmień nazwę pliku na *vowels4.py*, a następnie dokonaj odpowiednich modyfikacji w nowej wersji tego programu.

Magnesiki do zliczania wystąpień. Rozwiązanie

Gdy podsumowaliśmy już wyniki naszych eksperymentów w powłoce, przyszedł czas, aby wprowadzić zmiany w programie *vowels3.py*. Twoim zadaniem było ułożenie magnesików w taki sposób, aby powstał z nich działający kod, który zlicza wystąpienia wskazanych samogłosek w podanym słowie.

Po umieszczeniu magnesików w miejscach, w których powinny się według Ciebie znaleźć, miałeś otworzyć kod programu *vowels3.py* w oknie edycji środowiska IDLE, zmienić nazwę pliku na *vowels4.py* i dokonać odpowiednich modyfikacji w nowej wersji tego programu.

To jest program vowels4.py.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
```

Utwórz pusty słownik.

```
found = {}
```

Zainicjalizuj wartość powiązaną z każdym kluczem (czyli każdą samogłoską) za pomocą zera.

```
found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0
```

```
for letter in word:
    if letter in vowels:
        found[letter] += 1
```

Zwiększ o jeden wartość, do której odnosi się found[letter].

Z uwagi na to, że w pętli for wykorzystana została funkcja items, trzeba zapewnić dwie zmienne pętli. Są to zmienna k dla klucza oraz v dla wartości.

```
for k, v in sorted( found.items() ):
    print( k , 'znaleziono', v , 'raz(y).')
```

Wywołaj metodę items na rzecz słownika found, aby w kolejnych iteracjach odwołać się do każdego wiersza danych.

Klucz i wartość są używane do tworzenia każdego z komunikatów wyjściowych.

Te magnesiki nie były nam potrzebne.

```
found
key
value
found = []
```

Jazda próbna

Weźmy program *vowels4.py* w obroty. Mając kod w oknie edycji środowiska IDLE, naciśnij klawisz *F5*, aby przekonać się, jak Twój program radzi sobie z zadaniem, które przed nim postawiliśmy.

To jest kod programu vowels4.py.

vowels4.py - C:/Users/lsuma/Desktop/Python/R02/vowels4.py (3.6.0)

File Edit Format Run Options Window Help

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
    if letter in vowels:
        found[letter] += 1
```

Uruchomiliśmy ten kod trzykrotnie, aby zobaczyć, jak dobrze radzi sobie z różnymi danymi wejściowymi.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels4.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: mizerni
a znaleziono 0 raz(y).
e znaleziono 1 raz(y).
i znaleziono 2 raz(y).
o znaleziono 0 raz(y).
u znaleziono 0 raz(y).
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels4.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: życie, wszechświat i cała reszta
a znaleziono 4 raz(y).
e znaleziono 3 raz(y).
i znaleziono 3 raz(y).
o znaleziono 0 raz(y).
u znaleziono 0 raz(y).
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R02/vowels4.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: hm
a znaleziono 0 raz(y).
e znaleziono 0 raz(y).
i znaleziono 0 raz(y).
o znaleziono 0 raz(y).
u znaleziono 0 raz(y).
>>>
```

Te trzy wykonania programu wygenerowały dane wyjściowe, które okazały się zgodne z naszymi oczekiwaniami.

Podoba mi się kierunek, w którym to wszystko zmierza. Ale czy naprawdę muszę być informowany, o każdej samogłosce, która nie została znaleziona?

Jak dynamiczne są słowniki?

Program *vowels4.py* wyświetla informacje na temat wszystkich samogłosek znajdujących się na wzorcowej liście, nawet gdy nie zostały one znalezione w przeszukiwanym słowie. Może Ci to w ogóle nie przeszkadzać, wyobraźmy sobie jednak, że jest inaczej i chcesz, aby nasz kod wyświetlał wyniki tylko dla tych samogłosek, które *rzeczywiście* udało się znaleźć. Chodzi nam po prostu o to, aby pozbyć się z ekranu wszystkich tych komunikatów mówiących, że daną samogłoskę „znaleziono 0 raz(y)".

Co mógłbyś zrobić, aby rozwiązać ten problem?

Słownik Pythona ma charakter dynamiczny, zgadza się? A zatem wszystko, co powinniśmy tu zrobić, to usunąć te pięć wierszy kodu, w których odbywała się inicjalizacja liczników wystąpień poszczególnych samogłosek? Gdy się ich pozbędziemy, zliczane będą jedynie odnalezione samogłoski, prawda?

To brzmi, jakby mogło zadziałać.

Obecnie mamy pięć wierszy kodu znajdujących się w początkowej części programu *vowels4.py*, które dodaliśmy, aby *wstępnie* ustawić licznik związany z każdą samogłoską na wartość 0. Powoduje to utworzenie pary klucz-wartość dla każdej samogłoski, nawet jeśli niektóre z nich nigdy nie zostaną użyte. Jeśli usuniemy te pięć wierszy kodu, powinniśmy doprowadzić do sytuacji, w której liczniki wystąpień będą miały wyłącznie te samogłoski, które udało się znaleźć, a pozostałe będą ignorowane.

Spróbujmy zastosować to rozwiązanie w praktyce.

To jest kod programu vowels5.py, w którym kod inicjalizujący został usunięty.

Zrób to sam!

Weź kod programu *vowels4.py* i zapisz go w pliku *vowels5.py*. Następnie usuń pięć wierszy kodu inicjalizującego. Twoje okno edycji środowiska IDLE powinno wyglądać jak to pokazane obok.

vowels5.py - C:\Users\lsuma\Desktop\Python\R03\vowels5.py (3.6.0)

File Edit Format Run Options Window Help

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'znaleziono', v, 'raz(y).')
```

Ln: 13 Col: 0

Jazda próbna

Znasz procedurę. Upewnij się, że w oknie edycji środowiska IDLE widać kod *vowels5.py*, a następnie naciśnij klawisz *F5*, aby uruchomić swój program. W oknie powłoki zobaczysz komunikat błędu czasu wykonania.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
========== RESTART: C:\Users\lsuma\Desktop\Python\R03\vowels5.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: mizerni
Traceback (most recent call last):
  File "C:\Users\lsuma\Desktop\Python\R03\vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
                                                          Ln: 78  Col: 4
```

To nie może być dobre rozwiązanie.

To już całkowicie jasne, że usunięcie pięciu wierszy kodu inicjalizującego nie było sposobem, który powinniśmy tu wybrać. Ale dlaczego do tego doszło? To, że słownik Pythona rozrasta się dynamicznie w czasie wykonania programu, powinno przecież oznaczać, iż kod ten *nie może* ulec awarii, a jednak tak się właśnie stało. Dlaczego dostaliśmy ten błąd?

Klucze słownika muszą zostać zainicjalizowane

Usunięcie kodu inicjalizacyjnego spowodowało wystąpienie błędu czasu wykonania, a dokładnie — błędu `KeyError`, który pojawia się, gdy próbujesz odwołać się do wartości związanej z nieistniejącym kluczem. Jako że klucz ten nie może zostać znaleziony, związanej z nim wartości również nie da się znaleźć i właśnie dlatego zgłaszany jest tu błąd.

Czy to oznacza, że musimy z powrotem wprowadzić kod inicjalizujący? W końcu to tylko pięć krótkich wierszy kodu, więc w czym problem? Z całą pewnością możemy to zrobić, ale spróbujmy to najpierw spokojnie przemyśleć.

Wyobraź sobie, że zamiast pięciu liczników wystąpień musisz śledzić ich tysiąc (lub jeszcze więcej). Oznacza to, że nagle mamy już *mnóstwo* kodu inicjalizującego. Moglibyśmy „zautomatyzować” inicjalizację za pomocą pętli, ale nadal musielibyśmy tworzyć ogromny słownik, zawierający mnóstwo wierszy, z których wiele nigdy nie zostałoby użytych.

Gdyby tylko istniał jakiś sposób tworzenia pary klucz-wartość w locie dopiero wtedy, gdy okazałoby się, że jest nam ona potrzebna.

Porady maniaka

Alternatywne podejście do rozwiązania tego problemu mogłoby polegać na obsłudze występującego tu wyjątku czasu wykonania (którym w naszym przypadku jest `KeyError`). Opis sposobów radzenia sobie z wyjątkami czasu wykonania w Pythonie został zamieszczony w jednym z kolejnych rozdziałów, dlatego na tym etapie nie będziemy się już więcej rozpisywać na ten temat, a Ciebie zachęcamy do kontynuowania lektury.

To doskonałe pytanie.

Operator `in` spotkałeś po raz pierwszy przy okazji sprawdzania, czy na liście znajduje się określona wartość. Być może operator ten działa również w odniesieniu do słowników?

Przekonajmy się o tym, eksperymentując z kodem w wierszu poleceń powłoki.

Unikanie wyjątków KeyError w czasie wykonania

Analogicznie jak w przypadku list, operator `in` można stosować w celu sprawdzenia, czy dany klucz znajduje się w słowniku; interpreter zwraca w takiej sytuacji wartość `True` lub `False` w zależności od tego, co udało się stwierdzić.

Skorzystajmy z tego, aby uniknąć wyjątku `KeyError`, ponieważ dość denerwujące może być obserwowanie, jak wykonywanie Twojego kodu się zatrzymuje w wyniku pojawienia się tego błędu przy próbie wypełnienia słownika danymi w czasie działania programu.

Aby zademonstrować tę technikę, utworzymy słownik o nazwie `fruits`, a następnie skorzystamy z operatora `in`, aby uniknąć zgłaszania wyjątku `KeyError` podczas próby uzyskania dostępu do nieistniejącego klucza. Zaczniemy od utworzenia pustego słownika, potem dodamy do niego parę klucz-wartość łączącą wartość 10 z kluczem `jabłka`. Mając ten wiersz danych w słowniku, będziemy mogli zastosować operator `in`, aby potwierdzić, że klucz `jabłka` znajduje się już w tej kolekcji, tak jak zostało to pokazane poniżej.

```
>>> fruits = {}
>>> fruits
{}
>>> fruits['jabłka'] = 10
>>> fruits
{'jabłka': 10}
>>> 'jabłka' in fruits
True
```

Wszystko odbywa się tu zgodnie z oczekiwaniami. Wartość jest powiązana z kluczem i nie pojawia się żaden błąd czasu wykonania, gdy korzystamy z operatora in, aby sprawdzić, czy klucz ten istnieje.

Zanim przejdziemy dalej, rozważmy, jak interpreter „widzi” słownik `fruits` w pamięci po wykonaniu kodu przedstawionego powyżej.

Klucz ‘jabłka’ jest powiązany z wartością 10.

Nie istnieją głupie pytania

P: Z przykładu zaprezentowanego na tej stronie wnioskuję, że w Pythonie korzysta się ze stałej wartości `True` do oznaczania wartości *prawda*, zgadza się? Czy istnieje w związku z tym również `False` i czy te wielkie litery mają znaczenie, gdy używa się obydwu tych wartości?

O: Odpowiedź na wszystkie pytania brzmi: tak. Gdy w Pythonie masz określić wartość logiczną, możesz użyć jednej z wartości: `True` lub `False`. Są to wartości stałe zapewniane przez interpreter i muszą być zapisywane wielką literą, ponieważ interpreter traktuje `true` oraz `false` jako nazwy zmiennych, a *nie* wartości boolowskie — zatem zdecydowanie musisz tu zachować ostrożność.

Sprawdzanie przynależności za pomocą operatora in

Do słownika `fruits` dodajmy jeszcze jeden wiersz danych — z kluczem `banany` — i zobaczmy, co się stanie. Jednak zamiast prostego przypisania (z którego korzystaliśmy w przypadku wiersza zawierającego klucz `jabłka`) zastosujmy tym razem inkrementację wartości związanej z kluczem `banany`, jeśli klucz ten już istnieje w słowniku `fruits`, a jeśli nie istnieje, zainicjalizujmy tę daną wartością 1. To bardzo typowa operacja, zwłaszcza w przypadku zliczania liczby wystąpień za pomocą słownika, a zastosowany tu sposób powinien nam pomóc uniknąć wystąpienia wyjątku `KeyError`.

Stan przed wykonaniem kodu dotyczącego bananów

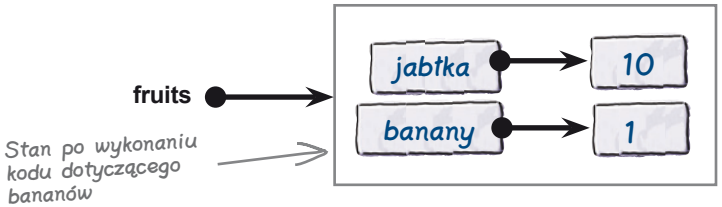
W przedstawionym poniżej kodzie operator `in` wraz z instrukcją `if` pozwalają nam uniknąć wszelkich potknięć związanych z dostępem do klucza `banany`, które to potknięcia — jak to zwykle bywa w przypadku tego rodzaju owoców — mogą się źle skończyć (nawet dla nas).

```
>>> if 'banany' in fruits:
    fruits['banany'] += 1
else:
    fruits['banany'] = 1

>>> fruits
{'jabłka': 10, 'banany': 1}
```

Sprawdzamy, czy klucz 'banany' znajduje się w słowniku, a ponieważ go tam nie ma, inicjalizujemy związaną z nim wartość za pomocą liczby 1. Unikamy w ten sposób jakiejkolwiek możliwości wystąpienia wyjątku KeyError.

Ustawiliśmy wartość związaną z kluczem 'banany' na 1.

Pokazany powyżej kod powoduje zmiany stanu słownika `fruits` w pamięci interpretera w sposób przedstawiony poniżej.



Stan po wykonaniu kodu dotyczącego bananów

Tak jak się spodziewaliśmy, słownik `fruits` został powiększony o jedną parę klucz-wartość, a wartość powiązana z kluczem `banany` została zainicjalizowana liczbą 1. Stało się tak, ponieważ warunek sprawdzany przez instrukcję `if` okazał się fałszywy (gdyż klucza nie udało się znaleźć w słowniku). Z tego powodu zamiast instrukcji inkrementacji wykonana została instrukcja przypisania, która znajduje się w drugim zestawie (związanym z częścią `else` instrukcji warunkowej). Zobaczmy, co się stanie, gdy kod ten zostanie wykonany raz jeszcze.

Porady maniaka

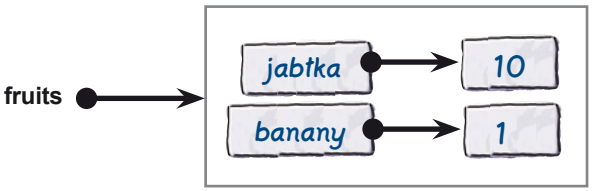
Jeśli z innych języków programowania znany Ci jest **trójargumentowy operator** `? :`, wiedz, że Python zapewnia możliwość korzystania z podobnej konstrukcji. Możesz napisać następujący wiersz kodu:

x = 10 if y > 3 else 20

aby zmiennej x przypisać wartość 10 lub 20 w zależności od tego, czy wartość zmiennej y jest większa niż 3. Musisz też jednak wiedzieć, że większość programistów korzystających z Pythona skrzywiłaby się z niesmakiem na widok zastosowania tej konstrukcji, ponieważ uważają oni, że odpowiadające jej znaczeniowo instrukcje `if... else...` znacznie łatwiej się czyta.

Zapewnianie inicjalizacji przed użyciem

Jeśli wykonamy ten kod ponownie, wartość powiązana z kluczem `banany` powinna zostać zwiększona o 1, ponieważ tym razem zostanie wykonany zestaw kodu związany z instrukcją `if`, a to dlatego, że klucz `banany` istnieje już w słowniku `fruits`, co pokazano poniżej.



Stan przed (ponownym) wykonaniem kodu dotyczącego bananów

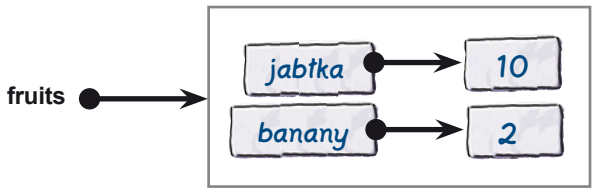
Aby uruchomić ten kod ponownie, naciśnij kombinację klawiszy *Alt+P* (jeśli korzystasz z systemu Windows lub Linux) albo *Ctrl+P* (jeśli używasz Maca) w celu przejrzenia historii i wybrania odpowiedniej instrukcji kodu spośród tych, które zostały ostatnio wprowadzone po znaku zachęty >>> widocznym w wierszu poleceń powłoki (jako że klawisz strzałki skierowanej w górę, standardowy w takich sytuacjach, nie umożliwia tutaj przywołania wpisywanych ostatnio poleceń). Pamiętaj, że aby uruchomić kod, musisz *dwukrotnie* nacisnąć klawisz *Enter*.

```
>>> if 'banany' in fruits:
    fruits['banany'] += 1
else:
    fruits['banany'] = 1

>>> fruits
{'jabłka': 10, 'banany': 2}
```

Tym razem klucz 'banany' istnieje już w słowniku, dlatego inkrementujemy powiązaną z nim wartość. Tak jak poprzednio, również i teraz tandem składający się z instrukcji if oraz operatora in zapobiegł awarii spowodowanej wyjątkiem KeyError.

Zwiększyliśmy wartość związaną z kluczem 'banany' o 1.

Tym razem wykonuje się kod związany z instrukcją `if`, dlatego wartość powiązana z kluczem `banany` zostaje zinkrementowana w pamięci interpretera.

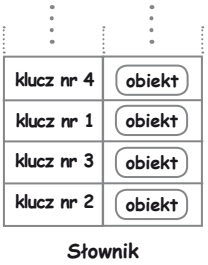


Po wykonaniu kodu dotyczącego bananów wartość powiązana z tym kluczem wzrosła.

Mechanizm ten jest tak często wykorzystywany, że wielu programistów korzystających z języka Python skraca przedstawione powyżej cztery wiersze kodu, zmieniając używany w nich warunek na przeciwny. Zamiast stosować operator `in`, używają oni połączenia operatorów `not in`. Umożliwia to zainicjalizowanie wartości klucza za pomocą pewnej wartości początkowej (zwykle jest to 0), gdy nie zostanie on znaleziony w słowniku, i wykonanie inkrementacji tuż po tym.

Przekonajmy się, jak ten mechanizm działa.

Zastępowanie operatora in operatorami not in



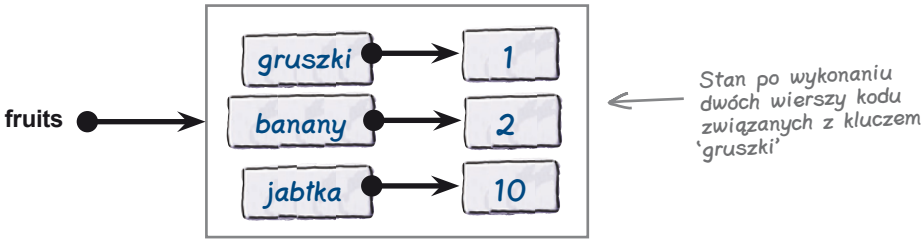
W dolnej części poprzedniej strony stwierdziliśmy, że większość programistów Pythona poddaje nasze cztery oryginalne wiersze kodu refaktoryzacji, w wyniku której zamiast operatora `in` stosowane jest połączenie operatorów `not it`. Zobaczmy to w akcji, korzystając z tego mechanizmu w celu zapewnienia, że wartość powiązana z kluczem `gruszki` jest ustawiona na 0, zanim zaczniemy ją zwiększać.

```
>>> if 'gruszki' not in fruits:
        fruits['gruszki'] = 0

>>> fruits['gruszki'] += 1
>>> fruits
{'jabłka': 10, 'banany': 2, 'gruszki': 1}
```

Inicjalizacja (jeśli jest konieczna)

Inkrementacja

Wykonanie przedstawionych powyżej trzech wierszy kodu spowodowało ponowne powiększenie słownika `fruits`. Znajdują się w nim teraz trzy pary klucz-wartość.



Trzy przedstawione powyżej wiersze kodu są powszechnie stosowane w Pythonie. Język ten zapewnia metodę słownikową, która sprawia, że to połączenie `if/not` staje się jeszcze bardziej wygodne w użyciu i jest obarczone mniejszym ryzykiem pojawienia się błędu. Metoda `setdefault` pozwala osiągnąć ten sam efekt, co zajmujące dwa wiersze instrukcje `if/not in`, ale mieści się w *pojedynczym* wierszu kodu.

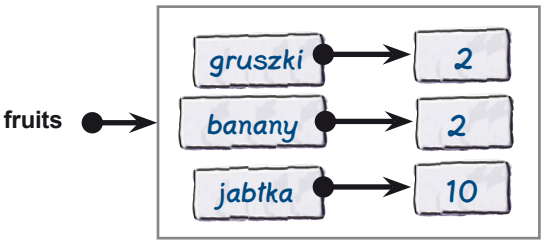
Poniżej znajdziesz napisany z wykorzystaniem metody `setdefault` odpowiednik kodu zaprezentowanego w górnej części strony, który to kod odpowiada za przetwarzanie danych związanych z gruszkami.

```
>>> fruits.setdefault('gruszki', 0)
>>> fruits['gruszki'] += 1
>>> fruits
{'jabłka': 10, 'banany': 2, 'gruszki': 2}
```

Inicjalizacja (jeśli jest konieczna)

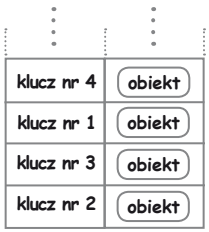
Inkrementacja

Pojedyncze wywołanie metody `setdefault` zastąpiło dwuwierszową konstrukcję `if/not in`. Jej zastosowanie zagwarantowało ponadto, że wartość związana z odpowiednim kluczem zawsze zostanie zainicjalizowana wartością początkową, zanim klucz ten zostanie użyty. Udało się w ten sposób wyeliminować możliwość pojawienia się wyjątku `KeyError`. Aktualny stan słownika `fruits` został pokazany obok, aby upewnić Cię, że wywołanie metody `setdefault` nie ma wpływu na dane (w tym przypadku na wartość powiązaną z kluczem `gruszki`), jeśli właściwy klucz już wcześniej istnieje. Sytuacja ta jest w pełni zgodna z naszymi oczekiwaniami.



Zaprzęganie metody setdefault do pracy

Przypomnij sobie, że nasza bieżąca wersja programu *vowels5.py* powoduje zgłoszenie błędu czasu wykonania, a dokładnie — wystąpienie wyjątku `KeyError`, który pojawia się, ponieważ nasz kod próbuje uzyskać dostęp do wartości związanej z nieistniejącym kluczem.



Słownik

Ten kod powoduje zgłoszenie następującego błędu.

vowels5.py - C:\Users\lsuma\Desktop\Python\R03\vowels5.py (3.6.0)

File Edit Format Run Options Window Help

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1
```

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
========== RESTART: C:\Users\lsuma\Desktop\Python\R03\vowels5.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: mizerni
Traceback (most recent call last):
  File "C:\Users\lsuma\Desktop\Python\R03\vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
```

Ln: 78 Col: 4

Dzięki naszym eksperymentom ze słownikiem `fruits` wiemy, że metodę `setdefault` możemy wywoływać tak często, jak tylko mamy ochotę, nie martwiąc się o to, że wystąpią z tego powodu jakieś brzydkie błędy. Wiemy też na pewno, że działanie tej metody polega na zainicjalizowaniu wartości powiązanej z nieistniejącym kluczem za pomocą dostarczonej wartości standardowej lub na nierobieniu zupełnie niczego (tj. pozostawieniu istniejącej wartości powiązanej z jakimkolwiek istniejącym kluczem), gdy klucz ten jest obecny w słowniku. Jeśli wywołamy metodę `setdefault` bezpośrednio przed próbą użycia klucza w kodzie naszego programu *vowels5.py*, mamy gwarancję uniknięcia wyjątku `KeyError` niezależnie od tego, czy odpowiedni klucz będzie w tym momencie istniał, czy też nie. W każdym z tych przypadków nasz program będzie nadal działał i nie ulegnie awarii (a wszystko to dzięki zastosowaniu metody `setdefault`).

Używaj metody setdefault, aby zabezpieczyć kod przed zgłoszeniem wyjątku KeyError.

W swoim oknie edycji środowiska IDLE zmień pierwszą pętlę `for` występującą w programie *vowels5.py* w taki sposób, aby wyglądała dokładnie tak jak ta, która została pokazana poniżej (a więc poprzez dodanie wywołania metody `setdefault`), a następnie zapisz nową wersję programu w pliku o nazwie *vowels6.py*.

```
for letter in word:
    if letter in vowels:
        found.setdefault(letter, 0)
        found[letter] += 1
```

Pojedynczy wiersz kodu może nieraz zmienić wszystko.

Jazda próbna

Mając najnowszą wersję programu z samogłoskami (czyli plik *vowels6.py*) otwartą w oknie edycji środowiska IDLE, naciśnij klawisz *F5*. Uruchom ją kilka razy, aby upewnić się, że wredny wyjątek `KeyError` już się więcej nie pojawia.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
========== RESTART: C:\Users\lsuma\Desktop\Python\R03\vowels6.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: autostopowicz
a znaleziono 1 raz(y).
i znaleziono 1 raz(y).
o znaleziono 3 raz(y).
u znaleziono 1 raz(y).
>>>
========== RESTART: C:\Users\lsuma\Desktop\Python\R03\vowels6.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: życie, wszechświat i cała reszta
a znaleziono 4 raz(y).
e znaleziono 3 raz(y).
i znaleziono 3 raz(y).
>>>
                                                                  Ln: 16  Col: 4
```

To wygląda już całkiem dobrze. Wyjątek KeyError został wyeliminowany.

Zastosowanie metody `setdefault` rozwiązało problem z wyjątkiem `KeyError`, z którym borykaliśmy się w tym kodzie. Użycie tej techniki umożliwia dynamiczne powiększanie słownika w czasie wykonania programu ze świadomością, że nowe pary klucz-wartość zostaną utworzone wyłącznie wtedy, gdy naprawdę będą potrzebne.

Jeśli będziesz w ten sposób korzystał z metody `setdefault`, **nigdy** nie będziesz musiał poświęcać czasu na wcześniejsze inicjalizowanie wszystkich wierszy tworzących Twój słownik.

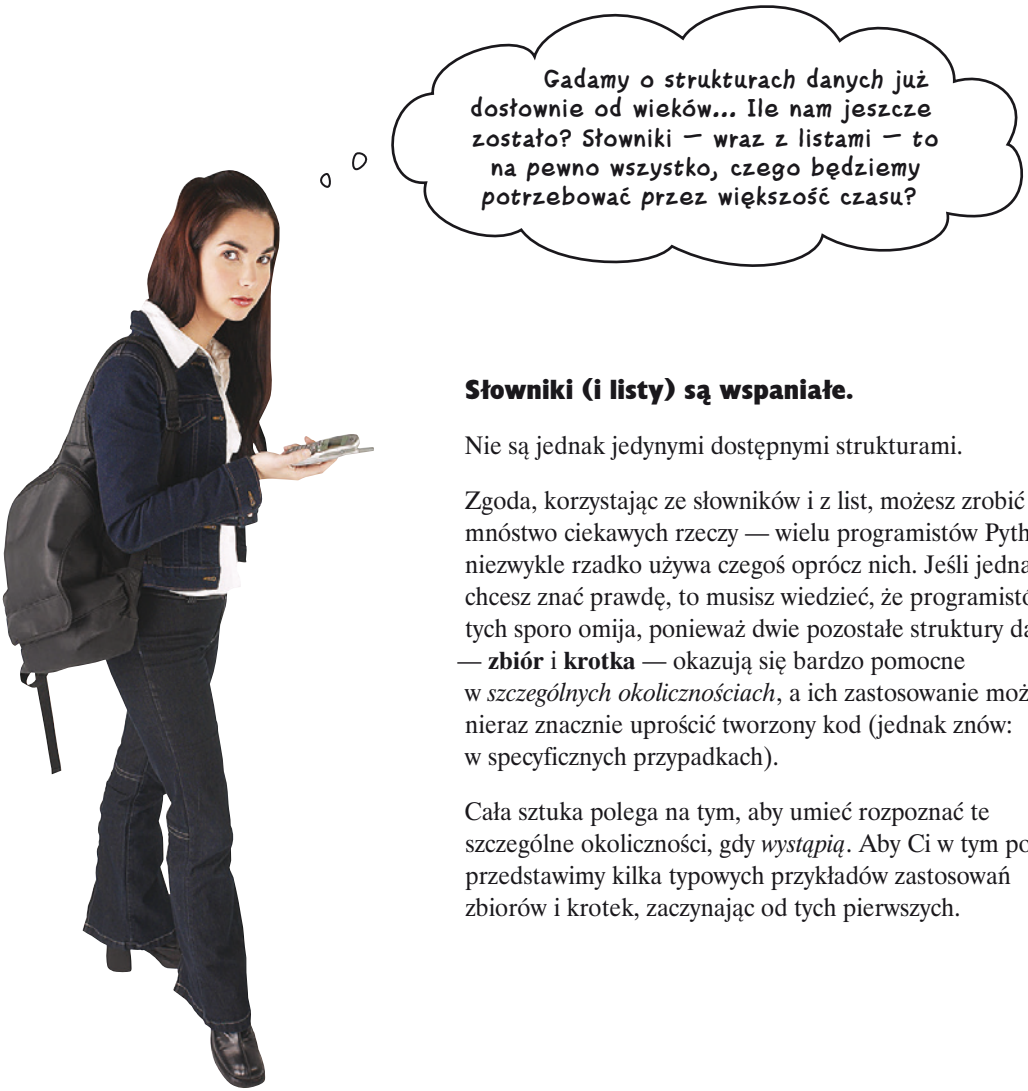
Słowniki: odświeżanie tego, co już wiemy

Do listy rzeczy, które wiesz już na temat słownika Pythona, dodaj punkty przedstawione poniżej.

CELNE SPOSTRZEŻENIA

- Standardowo każdy słownik jest nieuporządkowany, a kolejność wstawiania danych nie zostaje w nim zachowana. Jeśli jest potrzeba posortowania przechowywanych w słowniku danych podczas wyświetlania, należy skorzystać z wbudowanej funkcji `sorted`.
- Metoda `items` umożliwia iterowanie po słowniku wiersz po wierszu, tj. po poszczególnych parach klucz-wartość. W każdej iteracji metoda ta zwraca pętli `for` kolejny klucz oraz powiązaną z nim wartość.
- Próba odwołania się do nieistniejącego klucza w istniejącym słowniku powoduje zgłoszenie wyjątku `KeyError`. Pojawienie się tego wyjątku skutkuje awarią programu spowodowaną błędem czasu wykonania.
- Wyjątku `KeyError` można uniknąć, zapewniając, że z każdym kluczem znajdującym się w słowniku powiązana jest odpowiednia wartość przed pierwszym odwołaniem się do tej danej. Choć operatory `in` oraz `not in` mogą pomóc rozwiązać ten problem, lepszym sposobem jest wykorzystanie metody `setdefault`.

Czy słowniki (i listy) nam nie wystarczą?



Słowniki (i listy) są wspaniałe.

Nie są jednak jedynymi dostępnymi strukturami.

Zgoda, korzystając ze słowników i z list, możesz zrobić mnóstwo ciekawych rzeczy — wielu programistów Pythona niezwykle rzadko używa czegoś oprócz nich. Jeśli jednak chcesz znać prawdę, to musisz wiedzieć, że programistów tych sporo omija, ponieważ dwie pozostałe struktury danych — **zbiór** i **krotka** — okazują się bardzo pomocne w *szczególnych okolicznościach*, a ich zastosowanie może nieraz znacznie uprościć tworzony kod (jednak znów: w specyficznych przypadkach).

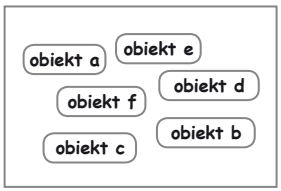
Cała sztuka polega na tym, aby umieć rozpoznać te szczególne okoliczności, gdy *wystąpią*. Aby Ci w tym pomóc, przedstawimy kilka typowych przykładów zastosowań zbiorów i krotek, zaczynając od tych pierwszych.

Nie istnieją głupie pytania

P: Czy to już wszystko na temat słowników? Z pewnością dość często zdarza się, że należąca do słownika wartość jest na przykład listą lub innym słownikiem, prawda?

O: Tak, to dość powszechny przypadek zastosowania słownika. Do końca tego rozdziału wstrzymamy się jednak z przedstawianiem Ci sposobu, w jaki można to zrobić. W międzyczasie postaraj się utrwalić w pamięci wszystko, co wiesz już na temat słowników...

Zbiory nie dopuszczają duplikatów



Zbiór

Zapewniana przez język Python struktura danych o nazwie **zbiór** bardzo przypomina matematyczny zbiór, o którym uczyłeś się w szkole. Ma określone, niezmienne właściwości matematyczne, a najważniejszą jego cechą jest to, że *zduplikowane wartości są w nim niedopuszczalne*.

Wyobraź sobie, że dostarczono Ci długą listę zawierającą wszystkie imiona pracowników ogromnego przedsiębiorstwa, interesuje Cię jednak wyłącznie (o wiele mniejsza) lista unikatowych imion. Potrzebujesz szybkiego i niezawodnego sposobu usunięcia wszystkich duplikatów z oryginalnej, długiej listy. W rozwiązywaniu tego rodzaju problemów doskonale sprawdzają się właśnie zbiory. Wystarczy, że skonwertujesz długą listę imion na postać zbioru (pozbywając się w ten sposób zbędnych powtórzeń), a następnie przekształcisz go z powrotem w listę, dzięki czemu — tadam! — otrzymasz potrzebną Ci listę unikatowych imion.

Zbiory w Pythonie zostały zoptymalizowane pod kątem bardzo szybkiego przeszukiwania, co sprawia, że ich zastosowanie jest o wiele wydajniejszym rozwiązaniem niż użycie odpowiadających im list, jeśli podstawowym wymaganiem jest krótki czas wyszukiwania danych. W takich przypadkach zastosowanie zbiorów zawsze powinno być rozważane na pierwszym miejscu, ponieważ listy umożliwiają jedynie przeprowadzanie powolnych, sekwencyjnych operacji wyszukiwania.

Znajdowanie zbiorów w kodzie

Zbiory można bardzo łatwo namierzyć w kodzie, ponieważ stanowią ujęte w nawiasy klamrowe kolekcje obiektów rozdzielonych przecinkami.

Poniżej został na przykład zaprezentowany pewien zbiór samogłosek.

Zbiory w kodzie zaczynają się od nawiasów klamrowych i na nich też się kończą.

```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'i', 'u', 'o', 'e', 'a'}
```

Sprawdź kolejność elementów. Została ona zmieniona w stosunku do oryginalnej kolejności wprowadzania danych, zniknęły też duplikaty.

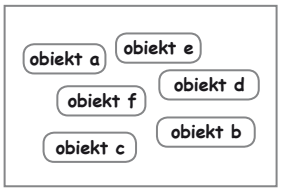
Poszczególne obiekty są od siebie oddzielone przecinkami.

To, że zbiór ujęty jest w nawiasy klamrowe, może nieraz spowodować, że pomylisz go ze słownikiem, który *również* zapisywany jest w takich nawiasach. Główną różnicą jest to, że w słownikach występuje znak dwukropka (`:`) oddzielający każdy klucz od powiązanej z nim wartości; znaku tego próżno szukać w zbiorach, występują tam wyłącznie przecinki.

Zauważ, że oprócz tego, że w zbiorach niedopuszczalne jest występowanie duplikatów, to również — tak jak ma to miejsce w przypadku słowników — kolejność wstawiania elementów do wykorzystywanego zbioru *nie* jest zachowywana przez interpreter. Jednak podobnie jak we wszystkich innych strukturach danych zbiory można porządkować przy wyprowadzaniu ich na wyjście za pomocą funkcji `sorted`. Ponadto, tak samo jak listy i słowniki, zbiory mogą rosnąć i kurczyć się zgodnie z potrzebami.

Struktura danych będąca zbiorem umożliwia przeprowadzanie typowych dla niego działań, takich jak *różnica*, *iloczyn* czy *suma* zbiorów. Aby zaprezentować Ci zbiory w akcji, raz jeszcze powrócimy do naszego przykładowego programu zliczającego wystąpienia samogłosek, który został przedstawiony we wcześniejszej części tego rozdziału. Podczas pracy nad programem *vowels3.py* (jeszcze w poprzednim rozdziale) obiecaliśmy Ci, że w roli podstawowej struktury danych zamiast listy spróbujemy w nim zastosować zbiór. Teraz nadszedł czas na spełnienie tej obietnicy.

Wydajne tworzenie zbiorów



Zbiór

Ponownie przyjrzyjmy się kodowi programu *vowels3.py*, w którym do sprawdzenia, czy samogłoski występują w zadanym słowie, wykorzystywana była lista.

Poniżej zaprezentowaliśmy ten kod raz jeszcze. Zauważ, że teraz mamy już w nim zaimplementowaną logikę, dzięki której każda samogłoska odnotowywana jest tylko raz. Z pełną premedytacją wyeliminowaliśmy tu *jakąkolwiek* możliwość dodania duplikatów do listy `found`.

To kod programu vowels3.py, który wyświetla unikatowe samogłoski znalezione w podanym słowie. W kodzie tym w roli podstawowej struktury danych została wykorzystana lista.

vowels3.py - C:/Users/lsuma/Desktop/Python/R02/vowels3.py (3.6.0)

File Edit Format Run Options Window Help

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

Nie dopuszczamy do pojawienia się duplikatów na liście found.

Zanim przejdziesz dalej, skorzystaj ze środowiska IDLE, aby zapisać ten kod w pliku o nazwie *vowels7.py*, dzięki czemu będziemy mogli swobodnie wprowadzać w nim zmiany, nie martwiąc się, że zepsujemy nasze rozwiązanie wykorzystujące w swoim działaniu listy (o którym wiemy, że funkcjonuje zgodnie z oczekiwaniami). Ponieważ stało się to już naszą rutynową praktyką, poeksperymentujmy w wierszu poleceń powłoki, zanim zabierzemy się do wprowadzania zmian w kodzie programu *vowels7.py*. Kod widoczny w oknie edycji środowiska IDLE zaczniemy edytować dopiero wtedy, gdy uda nam się już wypracować odpowiednie rozwiązanie w powłoce.

Tworzenie zbiorów z sekwencji

Zaczniemy od utworzenia zbioru samogłosek za pomocą kodu przedstawionego w połowie poprzedniej strony (krok ten możesz pominąć, jeśli wpisałeś już odpowiednią instrukcję po znaku zachęty >>> wiersza poleceń powłoki).

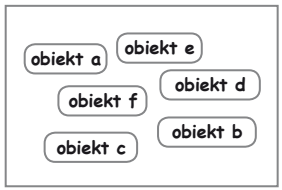
```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'i', 'u', 'o', 'e', 'a'}
```

Poniżej zaprezentowany został wygodny sposób osiągnięcia tego samego efektu. Umożliwia on szybkie wygenerowanie zbioru za pomocą funkcji `set`, której przekazuje się dowolną sekwencję (na przykład łańcuch znakowy). Sposób ten został pokazany na przykładzie tworzenia zbioru samogłosek.

Te dwa wiersze kodu spełniają to samo zadanie: każdy z nich przypisuje nowy obiekt zbioru do zmiennej.

```
>>> vowels2 = set('aeeiouu')
>>> vowels2
{'i', 'u', 'o', 'e', 'a'}
```

Korzystanie z metod zbioru



Zbiór

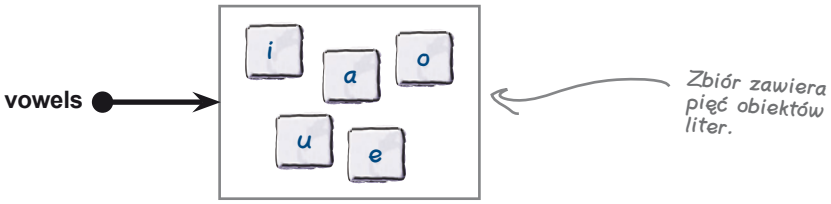
Teraz, gdy nasze samogłoski znajdują się już w zbiorze, kolejnym krokiem powinno być określenie, czy któreś ze znaków tworzących słowo należą do zbioru samogłosek. Moglibyśmy to zrobić, sprawdzając, czy poszczególne litery, z których składa się dane słowo, znajdują się w tym zbiorze, jako że operator `in` działa w odniesieniu do zbiorów mniej więcej w ten sam sposób, w jaki robi to w odniesieniu do słowników i list. Moglibyśmy zatem skorzystać z `in`, aby sprawdzić, czy zbiór zawiera jakąkolwiek literę, a następnie za pomocą pętli `for` cyklicznie sprawdzać występowanie poszczególnych liter słowa w zbiorze.

Jednak nie podążajmy tutaj tą ścieżką, ponieważ zastosowanie metod zbioru pomoże nam uniknąć większości czynności wykonywanych w pętlach, o których była mowa powyżej.

Istnieje bowiem o wiele lepszy sposób wykonywania tego rodzaju operacji za pomocą zbiorów. Wymaga on użycia metod, w które wyposażony jest każdy zbiór i które umożliwiają przeprowadzanie działań takich jak suma, różnica i iloczyn zbiorów. Zanim zaczniemy modyfikować kod programu *vowels7.py*, dowiedzmy się, jak działają te metody, eksperymentując w wierszu poleceń powłoki i zastanawiając się, jak interpreter „widzi” zbiory danych. Postaraj się wykonywać kolejne kroki na swoim komputerze. Zacznijmy od utworzenia zbioru samogłosek, po czym przypiszmy wartość do zmiennej `word`.

```
>>> vowels = set('aeiou')
>>> word = 'halo'
```

Interpreter tworzy dwa obiekty: jeden zbiór i jeden łańcuch znakowy. Poniżej zaprezentowano, jak zbiór `vowels` wygląda w pamięci interpretera.



Przekonajmy się, co się stanie, gdy wykonamy działanie sumy zbiorów na zbiorze `vowels` oraz zbiorze liter utworzonym z wartości przechowywanej w zmiennej `word`. Drugi z tych zbiorów utworzymy w locie, przekazując zmienną `word` wywołaniu funkcji `set`, której wynik działania zostanie następnie przekazany metodzie `union` wywoływanej na rzecz obiektu `vowels`. Wynikiem działania tej ostatniej będzie kolejny zbiór, który przypiszemy do kolejnej zmiennej (o nazwie `u`). Ta nowa zmienna będzie zawierała *kombinację* obiektów należących do obydwu wyjściowych zbiorów (czyli ich sumę).

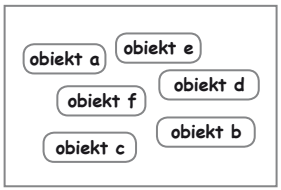
```
>>> u = vowels.union(set(word))
```

Python konwertuje wartość zapisaną w zmiennej word na zbiór obiektów liter (usuwając przy tym wszelkie duplikaty).

Metoda union łączy jeden zbiór z drugim, a wynik tej operacji jest następnie przypisywany do zmiennej o nazwie „u” (która jest kolejnym zbiorem).

Jak wyglądają zbiory `vowels` oraz `u` po tym wywołaniu metody `union`?

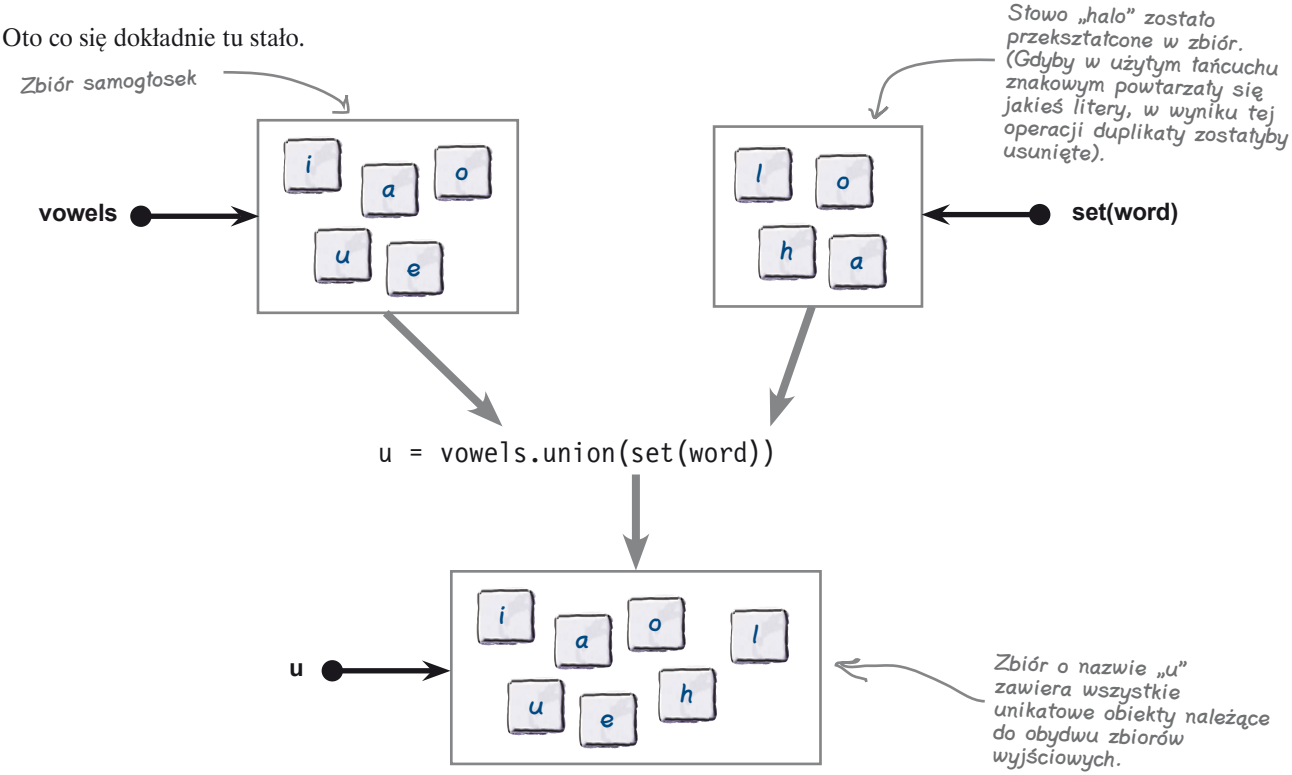
Działanie metody union polega na łączeniu zbiorów



Zbiór

Na końcu poprzedniej strony został pokazany przykład użycia metody `union` do tworzenia nowego zbioru o nazwie `u`, który był kombinacją liter należących do zbioru `vowels` oraz zbioru unikatowych liter tworzących słowo zapisane w zmiennej `word`. Operacja ta nie miała żadnego wpływu na zbiór `vowels`, który zachował swój stan sprzed wywołania operacji łączenia zbiorów. Zbiór `u` jest jednak zupełnie nowy, ponieważ powstał w wyniku działania metody `union`.

Oto co się dokładnie tu stało.



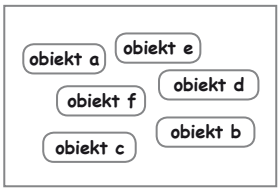
Co się stało z kodem pętli?

Ten pojedynczy wiersz kodu odwala kawał dobrej roboty. Zauważ, że nie kazałeś tu nigdzie interpreterowi wykonywać instrukcji pętli. Zamiast tego określiłeś tylko, *co* chciałeś, aby zostało zrobione, nie zaś *jak* miało się to odbyć, a interpreter spełnił Twoje życzenie, tworząc nowy zbiór zawierający obiekty, o które Ci chodziło.

Typowym wymaganiem (gdy utworzyliśmy już sumę zbiorów) jest przekształcenie otrzymanego zbioru w posortowaną listę. Operacja ta jest bardzo prosta do wykonania dzięki zastosowaniu funkcji `sorted` oraz `list`.

```
>>> u_list = sorted(list(u))
>>> u_list
['a', 'e', 'h', 'i', 'l', 'o', 'u']
```

Metoda difference informuje o elementach, które nie są wspólne



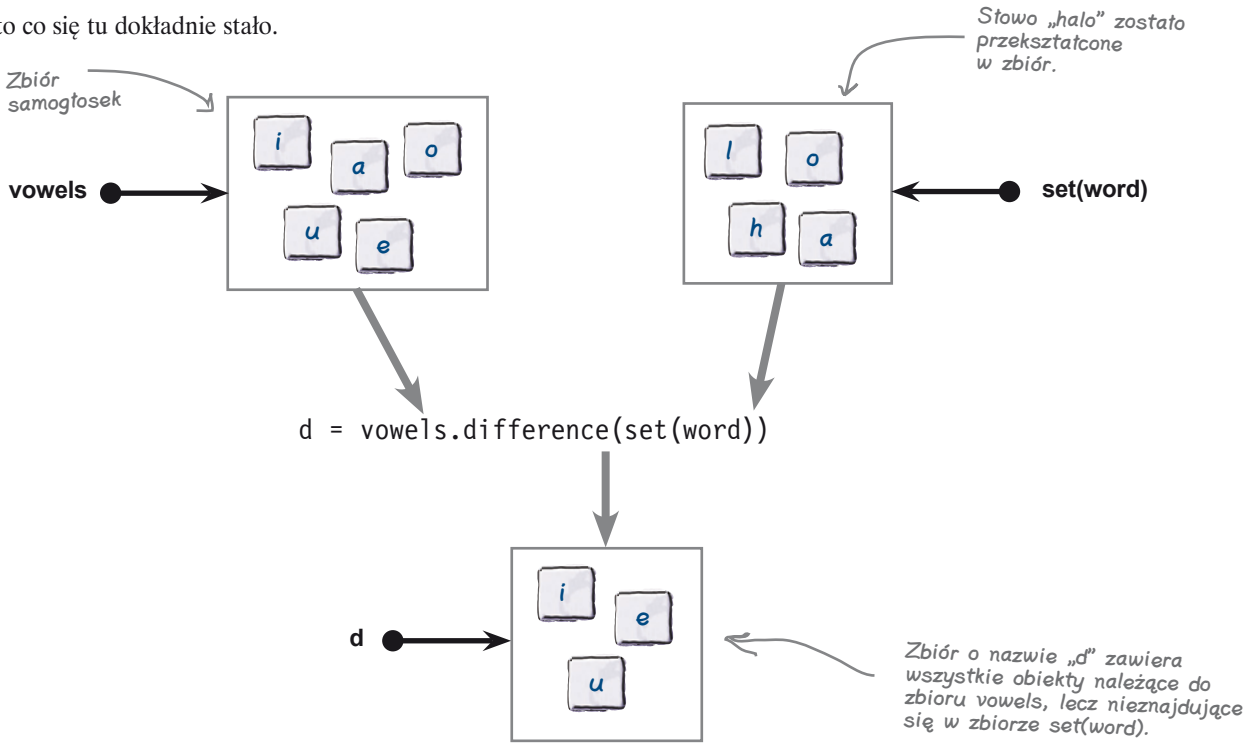
Zbiór

Inną metodą zbioru jest `difference`. Gdy otrzyma ona dwa zbiory, informuje, które elementy należą do jednego zbioru, nie występując jednak w drugim. Skorzystajmy z tej metody w sposób bardzo podobny do tego, jakiego używaliśmy w przypadku metody `union`, a następnie sprawdźmy, jaki wynik udało się nam uzyskać.

```
>>> d = vowels.difference(set(word))
>>> d
{'i', 'u', 'e'}
```

Przedstawione powyżej wywołanie metody `difference` porównuje obiekty znajdujące się w zbiorze `vowels` z obiektami należącymi do zbioru `set(word)`, a następnie zwraca nowy zbiór obiektów (noszący w naszym przykładzie nazwę `d`), które należą do zbioru `vowels`, lecz *nie* należą do zbioru `set(word)`.

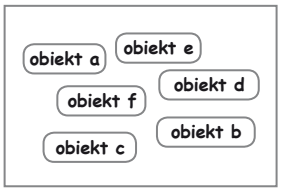
Oto co się tu dokładnie stało.



Jeszcze raz zwrócimy Twoją uwagę na fakt, że wynik ten udało się uzyskać *bez* korzystania z pętli `for`. Metoda `difference` wykonuje tu całą brudną robotę; nasze zadanie sprowadziło się do określenia tego, czego potrzebowaliśmy.

Przejdź do następnej strony, aby zapoznać się z ostatnią interesującą nas metodą zbioru, czyli metodą `intersection`.

Metoda intersection informuje o części wspólnej



Zbiór

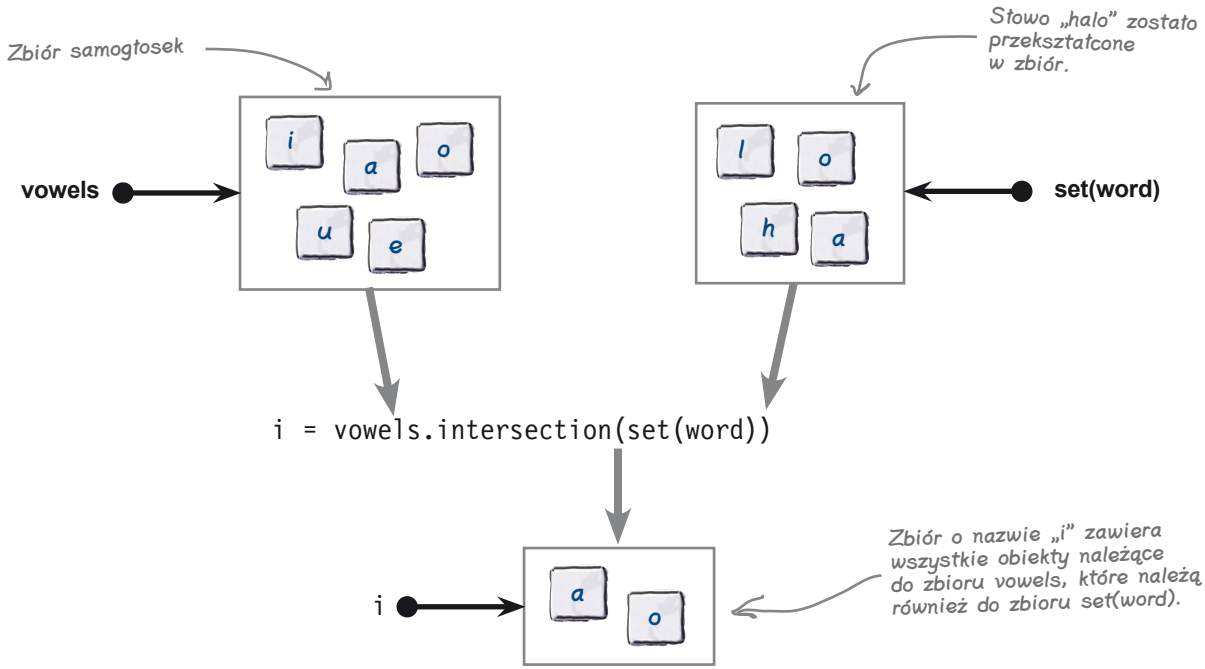
Trzecią metodą zbioru, jakiej się przyjrzymy, jest metoda `intersection`, która bierze obiekty należące do jednego zbioru i porównuje je do obiektów znajdujących się w drugim zbiorze, a następnie informuje o wszystkich wspólnych obiektach, które udało się znaleźć.

W kontekście tego, co mamy osiągnąć za pomocą programu *vowels7.py*, sposób działania metody `intersection` wydaje się bardzo obiecujący, ponieważ chcemy tu sprawdzić, które litery w podanym słowie są samogłoskami należącymi do zdefiniowanego zbioru.

Przypomnij sobie, że zmienna `word` przechowuje łańcuch znakowy `"halo"`, a nasze samogłoski znajdują się w zbiorze o nazwie `vowels`. Poniżej możesz zobaczyć metodę `intersection` w akcji.

```
>>> i = vowels.intersection(set(word))
>>> i
{'o', 'a'}
```

Metoda `intersection` potwierdza, że w słowie zapisanym w zmiennej `word` występują samogłoski `o` i `a`. Oto co się tu dokładnie wydarzyło.



Istnieje więcej metod zbioru niż trzy, które przedstawiliśmy na kilku ostatnich stronach, ale z tych trzech najbardziej interesująca dla nas jest tu metoda `intersection`. Za pomocą jednego wiersza kodu udało nam się rozwiązać problem, który postawiliśmy niemal na samym początku poprzedniego rozdziału, polegający na *identyfikacji wskazanych samogłosek w dowolnym łańcuchu znakowym*. A wszystko to bez używania jakiejkolwiek pętli. Wróćmy do programu *vowels7.py* i zastosujmy w praktyce wiedzę, którą właśnie zdobyliśmy.

Zbiory: co już wiemy

Poniżej znajdziesz krótki przegląd wiadomości, które udało Ci się zgromadzić na temat pythonowej struktury danych, jaką jest zbiór.

CELNE SPOSTRZEŻENIA

- Zbiory w Pythonie nie dopuszczają duplikatów.
- Podobnie jak słowniki, zbiory umieszcza się w nawiasach klamrowych, jednak w przypadku zbiorów nie występują pary klucz-wartość. Każdy unikatowy obiekt należący do zbioru jest oddzielony od kolejnego obiektu przecinkiem.
- Również podobnie jak słowniki, zbiory nie zachowują porządku wprowadzania danych (można je jednak porządkować za pomocą funkcji `sorted`).
- Funkcji `set` można przekazać dowolną sekwencję, aby z należących do niej obiektów utworzyć odpowiedni zbiór elementów (pozbawiony wszelkich duplikatów).
- Zbiory oferują mnóstwo wbudowanych możliwości, w tym metody pozwalające wykonywać działania sumy, różnicy oraz iloczynu zbiorów.

Zaostrz ołówek

Poniżej raz jeszcze został przedstawiony kod programu *vowels3.py*.

Na podstawie wiedzy, którą właśnie zdobyłeś, wykreśl te fragmenty kodu, które nie są Ci już potrzebne. W pustym miejscu znajdującym się po prawej stronie napisz kod, który uważasz za konieczny, aby zmienić ten program wykorzystujący listy w program korzystający z możliwości oferowanych przez zbiory.

Podpowiedź: na koniec Twój kod powinien być o wiele krótszy.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy
wyszukać samogłoski: ")
found = []
for letter in word:
   if letter in vowels:
      if letter not in found:
         found.append(letter)
for vowel in found:
   print(vowel)
```

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Po zakończeniu pracy zapisz swój nowy program w pliku *vowels7.py*.

Zaostrz ołówek
Rozwiązanie

Poniżej raz jeszcze został przedstawiony kod programu *vowels3.py*.

Na podstawie wiedzy, którą wcześniej zdobyłeś, miałeś wykreślić te fragmenty kodu, które nie były Ci już potrzebne. W pustym miejscu znajdującym się po prawej stronie miałeś napisać kod, który uważałeś za konieczny, aby zmienić ten program wykorzystujący listy w program korzystający z możliwości oferowanych przez zbiory.

Podpowiedź: na koniec Twój kod powinien być o wiele krótszy.

Mnóstwa kodu trzeba się stąd pozbyć.

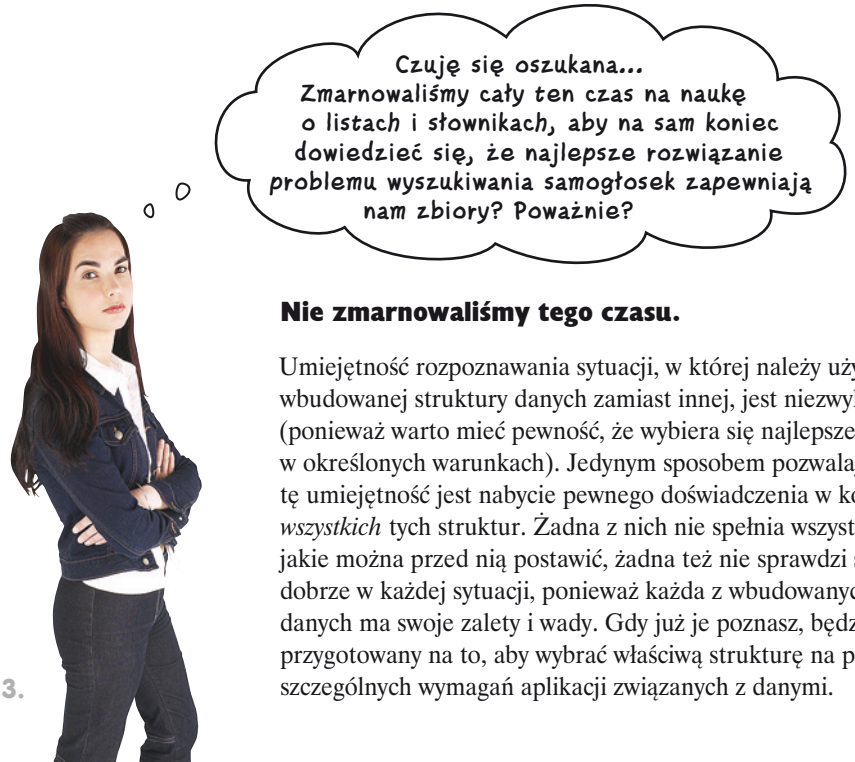
```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski:
")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Utworzenie zbioru samogłosek

```
vowels = set('aeiou')

found = vowels.intersection(set(word))
```

Te pięć wierszy kodu odpowiedzialnego za przetwarzanie listy zostało zastąpionych pojedynczym wierszem kodu operującego na zbiorach.

Po zakończeniu pracy miałeś zapisać swój nowy program w pliku *vowels7.py*.



Nie zmarnowaliśmy tego czasu.

Umiejętność rozpoznawania sytuacji, w której należy użyć jakiejś wbudowanej struktury danych zamiast innej, jest niezwykle istotna (ponieważ warto mieć pewność, że wybiera się najlepsze rozwiązanie w określonych warunkach). Jedynym sposobem pozwalającym zdobyć tę umiejętność jest nabycie pewnego doświadczenia w korzystaniu ze *wszystkich* tych struktur. Żadna z nich nie spełnia wszystkich wymagań, jakie można przed nią postawić, żadna też nie sprawdzi się równie dobrze w każdej sytuacji, ponieważ każda z wbudowanych struktur danych ma swoje zalety i wady. Gdy już je poznasz, będziesz lepiej przygotowany na to, aby wybrać właściwą strukturę na podstawie szczególnych wymagań aplikacji związanych z danymi.

Jazda próbna

Weźmy program *vowels7.py* w obroty, aby upewnić się, że rozwiązanie wykorzystujące zbiory działa równie dobrze jak starsza wersja, w której używane były listy.

Nasz najnowszy kod

vowels7.py - C:/Users/lsuma/Desktop/Python/R03/vowels7.py (3.6.0)

File Edit Format Run Options Window Help

```
vowels = set('aeiou')
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
found = vowels.intersection(set(word))
```

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R03/vowels7.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: niemy
e
i
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R03/vowels7.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: galaktyka
a
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R03/vowels7.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: życie, wszechświat i cała reszta
e
a
i
>>>
========== RESTART: C:/Users/lsuma/Desktop/Python/R03/vowels7.py ==========
Podaj słowo, w którym należy wyszukać samogłoski: hm
>>>
```

Ln: 21 Col: 4

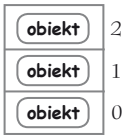
Wszystko działa zgodnie z założeniem.

Zastosowanie zbioru było tu doskonałym wyborem...

Nie oznacza to jednak, że dwie pozostałe struktury danych, które już poznałeś, nie mają swoich zastosowań. Jeśli musisz na przykład zliczać liczby wystąpień, oferowany przez język Python słownik sprawdzi się najlepiej. Jeżeli jednak bardziej zależy Ci na zachowaniu kolejności wprowadzania danych, wówczas odpowiednia okaże się wyłącznie lista... Co prawie jest prawdą. Istnieje inna wbudowana struktura, która zachowuje kolejność wprowadzania danych, a której jeszcze nie zdążyliśmy omówić, a mianowicie **krotka**.

Pozostałą część tego rozdziału spędźmy zatem w towarzystwie pythonowej krotki.

Tworzenie przypadku dla krotek



Krotka

Gdy programiści, dla których Python jest nowym językiem programowania, napotykają na swojej drodze **krotkę**, zadają sobie pytanie, po co taka struktura danych w ogóle istnieje. W końcu krotka to taka lista, której nie da się zmienić po utworzeniu (i zapełnieniu danymi). Krotki są niemodyfikowalne, a więc *nie mogą się zmieniać*. Do czego więc miałyby nam być potrzebne?

Okazuje się, że dysponowanie taką niemodyfikowalną strukturą danych może być nieraz bardzo przydatne. Wyobraź sobie, że musisz zabezpieczyć się przed jakimiś efektami ubocznymi, zapewniając, aby pewne dane w Twoim programie nigdy nie uległy zmianie. Albo masz do czynienia z długą stałą listą (o której wiesz, że nie ma prawa się zmienić), a bardzo zależy Ci na wysokiej wydajności. Po co ponosić koszty związane z całym tym dodatkowym kodem umożliwiającym przetwarzanie zwykłych (tj. modyfikowalnych) list, skoro nigdy nie będziesz go potrzebował? Zastosowanie krotki w takim przypadku pozwala uniknąć zbędnego narzutu i zabezpiecza Cię przed przykrymi efektami ubocznymi związanymi z danymi (gdyby miały one wystąpić).

Nie istnieją głupie pytania

P: Skąd wzięła się nazwa „krotka"?

O: To zależy, kogo zapytasz, ale nazwa ta ma swoje korzenie w matematyce. Więcej informacji na ten temat znajdziesz pod adresem *https://pl.wikipedia.org/wiki/Para_uporz%C4%85dkowana#Tr.C3.B3jki.2C_czw.C3.B3rki.2C_.E2.80.A6.2C_n-ki_uporz.C4.85dkowane.*

Jak znaleźć krotkę w kodzie

Jako że krotki są bardzo blisko spokrewnione z listami, nie jest wielkim zaskoczeniem, że wyglądają dość podobnie (i zachowują się w podobny sposób). Krotki umieszcza się w nawiasach okrągłych, a listy — w kwadratowych. Krótkie odwiedziny w wierszu poleceń powłoki umożliwią nam porównanie tych dwóch rodzajów struktur danych. Zwróć uwagę na to, jak korzystamy z wbudowanej funkcji `type`, aby potwierdzić typ każdego z utworzonych obiektów.

```
>>> vowels = [ 'a', 'e', 'i', 'o', 'u' ]
>>> type(vowels)
<class 'list'>
>>> vowels2 = ( 'a', 'e', 'i', 'o', 'u' )
>>> type(vowels2)
<class 'tuple'>
```

Nie ma tu nic nowego; tworzona jest lista samogłosek.

Wbudowana funkcja type zwraca informację na temat typu dowolnego obiektu.

Ta krotka wygląda jak lista, ale nią nie jest. Krotki umieszcza się w nawiasach okrągłych (a nie kwadratowych).

Teraz, gdy istnieją już obiekty `vowels` i `vowels2` (i są wypełnione danymi), możemy poprosić powłokę o wyświetlenie ich zawartości. Gdy to zrobimy, będziemy mieli potwierdzenie, że krotka nie jest dokładnie tym samym, co lista, tak jak zostało to pokazane poniżej.

```
>>> vowels
['a', 'e', 'i', 'o', 'u']
>>> vowels2
('a', 'e', 'i', 'o', 'u')
```

Nawiasy okrągłe wskazują, że mamy do czynienia z krotką.

Co się jednak stanie, gdy spróbujemy zmodyfikować krotkę?

Krotki są niezmienne

Krotka

Ponieważ krotki bardzo przypominają listy, zapewniają również obsługę tej samej notacji nawiasów kwadratowych, która kojarzona jest zwykle z listami. Wiemy już, jak skorzystać z tej notacji, aby zmienić zawartość listy. Poniżej pokazano, co zrobilibyśmy, gdybyśmy chcieli zastąpić przechowywaną na liście `vowels` małą literę `i` wielką literą, czyli `I`.

```
>>> vowels[2] = 'I'
>>> vowels
['a', 'e', 'I', 'o', 'u']
```

Przypisanie wielkiej litery „I" do trzeciego elementu listy vowels

Zgodnie z naszymi oczekiwaniami trzeci element listy (wskazywany przez indeks 2) uległ zmianie, co jest jak najbardziej w porządku i czego należało się spodziewać, ponieważ listy są strukturami modyfikowalnymi. Spójrz jednak, co się stanie, gdy tę samą operację spróbujemy przeprowadzić w stosunku do krotki `vowels2`.

Interpreter głośno protestuje, gdy próbujesz zmodyfikować krotkę.

```
>>> vowels2[2] = 'I'
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    vowels2[2] = 'I'
TypeError: 'tuple' object does not support item assignment
>>> vowels2
('a', 'e', 'i', 'o', 'u')
```

Brak jakichkolwiek zmian, ponieważ krotki są niemodyfikowalne.

Krotki są niemodyfikowalne, dlatego nie możemy narzekać, gdy interpreter protestuje w odpowiedzi na podjętą przez nas próbę zmiany obiektu przechowywanego w krotce. W końcu o to właśnie chodzi w krotkach: gdy zostały już utworzone i zapełnione danymi, nie powinno dać się ich zmienić.

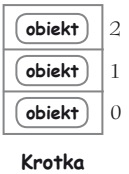
Nie popełniaj tu błędu: to zachowanie krotek jest pożądane, szczególnie gdy musisz zapewnić, aby pewnych danych pod żadnym pozorem nie dało się zmienić. Jedynym sposobem, który to umożliwia, jest umieszczenie danych w krotce, co stanowi dla interpretera jasną instrukcję, że ma powstrzymać każdy kod, który będzie próbował modyfikować dane krotki.

W dalszej części tej książki będziemy korzystać z krotek zawsze wtedy, gdy będzie to miało uzasadnienie. Wracając zaś do naszego kodu z samogłoskami, możemy już chyba z całą pewnością stwierdzić, że struktura danych `vowels` powinna być zawsze przechowywana w postaci krotki zamiast w postaci listy, ponieważ po prostu nie ma sensu używać w jej przypadku kolekcji modyfikowalnej (jako że samogłoski w naszym programie *nigdy* nie muszą się zmieniać).

Na temat krotek nie da się napisać właściwie nic więcej. Traktuj je po prostu jak niemodyfikowalne listy, nic ponadto. Struktury te mają jednak pewne zastosowanie, które stanowi spory problem dla wielu programistów, dlatego powinieneś dowiedzieć się, jak unikać potknięć tego rodzaju.

Jeśli dane przechowywane w strukturze nigdy się nie zmieniają, umieść je w krotce.

Uważaj na krotki składające się z pojedynczych obiektów



Wyobraźmy sobie, że w krotce chcemy przechowywać pojedynczy łańcuch znakowy. Aż kusi, aby umieścić go w nawiasach okrągłych, a następnie przypisać do jakiejś zmiennej. Jednak robiąc to, nie osiągniemy efektu, który spodziewamy się uzyskać.

Spójrz na zamieszczony tu eksperyment, który wykonaliśmy w wierszu poleceń powłoki. Pokazuje on, co się stanie, gdy spróbujesz wykonać opisaną powyżej operację.

```
>>> t = ('Python')
>>> type(t)
<class 'str'>
>>> t
'Python'
```

To nie to, czego się spodziewaliśmy. Mamy tu łańcuch znakowy (string). Co się stało z naszą krotką?

Coś, co wyglądało na krotkę zawierającą pojedynczy obiekt, bynajmniej nią nie jest; jest za to zwykłym łańcuchem znakowym. Stało się tak z powodu pewnego dziwactwa syntaktycznego występującego w języku Python. Regułą jest, że struktura, która ma być krotką, musi zawierać co najmniej jeden przecinek umieszczony w obrębie nawiasu okrągłego nawet wówczas, gdy mieści ona tylko jeden obiekt. W praktyce oznacza to, że gdy chcemy przypisać do krotki pojedynczy obiekt (którym w tym przypadku jest obiekt tekstowy), musimy dodać po nim znak przecinka.

```
t2 = ('Python',)
```

Ten końcowy przecinek zmienia tu dosłownie wszystko, ponieważ informuje interpreter, że ma do czynienia z krotką.

Wygląda to dość dziwnie, ale niech Cię to nie martwi. Po prostu zapamiętaj tę regułę, a wszystko będzie dobrze: *każda krotka musi zawierać przynajmniej jeden przecinek w nawiasie*. Gdy zapytasz teraz interpreter o typ (i zawartość) zmiennej `t2`, przekonasz się, że jest ona krotką, a więc dokładnie tym, czego się spodziewaliśmy.

```
>>> type(t2)
<class 'tuple'>
>>> t2
('Python',)
```

Tu już jest lepiej: teraz mamy krotkę.

Interpreter wyświetla krotkę zawierającą pojedynczy obiekt wraz z końcowym przecinkiem.

Wiele funkcji przyjmuje krotki jako argumenty wywołania, a także zwraca je jako wyniki swojego działania; jest tak nawet wtedy, gdy funkcje te przyjmują i zwracają pojedyncze obiekty. W związku z tym, korzystając z funkcji, często jesteś zmuszony stosować tę składnię. Kwestię związków występujących pomiędzy funkcjami i krotkami poruszymy jeszcze w dalszej części książki. W gruncie rzeczy funkcjom poświęcimy cały następny rozdział (dlatego nie będziesz musiał długo na to czekać).

Teraz, gdy znasz już cztery wbudowane struktury danych Pythona, a jeszcze przed tym, zanim przejdziemy do rozdziału traktującego o funkcjach, odejdźmy nieco od głównego wątku i przyjrzyjmy się krótkiemu — a także zabawnemu! — przykładowi wykorzystania bardziej skomplikowanej struktury danych.

Łączenie wbudowanych struktur danych

Cała ta gadka o strukturach danych spowodowała, że zacząłem się zastanawiać, czy sprawy mogą się jeszcze bardziej skomplikować. A konkretnie: czy w słowniku mogę przechowywać inny słownik?

To bardzo często zadawane pytanie.

Gdy programiści przyzwyczają się już do przechowywania na listach i w słownikach liczb, łańcuchów znakowych i wartości logicznych, szybko przechodzą do rozważań na temat tego, czy wbudowane struktury umożliwiają zapisywanie bardziej złożonych danych. A więc czy wbudowane struktury danych mogą same zawierać wbudowane struktury danych?

Odpowiedź brzmi: **tak**, a powód, dla którego tak się dzieje, jest taki, że *wszystko w Pythonie jest obiektem*.

Wszystko, co przechowywaliśmy do tej pory w każdej z wbudowanych struktur danych, było pewnego rodzaju obiektem. Fakt, że były to „proste obiekty" (takie jak liczby i łańcuchy znakowe) nie ma najmniejszego znaczenia, jako że wbudowane struktury danych są w stanie przechowywać *dowolne* obiekty. Wszystkie wbudowane struktury (pomimo że są „złożone") są także obiektami, dlatego możesz tu dowolnie dobierać to, co Ci najbardziej pasuje. Po prostu przypisz wbudowaną strukturę danych tak, jak zrobiłbyś to w przypadku prostego obiektu, i sprawa załatwiona.

Przyjrzyjmy się teraz przykładowi, w którym został zastosowany słownik słowników.

Nie istnieją głupie pytania

P: Czy to, co zamierzasz właśnie zrobić, sprawdza się wyłącznie w przypadku słowników? Czy też można mieć listę list, zbiór list lub krotkę słowników?

O: Tak, można. Pokażemy tu, jak działa słownik słowników, ale wbudowane struktury danych możesz łączyć w taki sposób, w jaki tylko zapragniesz.

Przechowywanie tabeli danych

Ponieważ wszystko jest obiektem, każda z wbudowanych struktur danych może być przechowywana w każdej wbudowanej strukturze danych, co umożliwia konstruowanie dowolnie skomplikowanych struktur danych... oczywiście przy założeniu, że Twój mózg poradzi sobie z wizualizowaniem sposobu jej działania. Na przykład, choć korzystanie ze *słownika list zawierających krotki, które z kolei zawierają zbiory słowników*, może Ci się wydawać doskonałym pomysłem, prawdopodobnie wcale takim nie jest, ponieważ złożoność całej tej konstrukcji wykracza poza skalę pojmowania przeciętnego programisty.

Struktura złożona, z którą w praktyce można się dość często spotkać, to słownik słowników. Struktura tego rodzaju może być wykorzystywana do tworzenia *tabeli modyfikowalnej* (ang. *mutable table*). Aby to zademonstrować, wyobraźmy sobie, że mamy przedstawioną poniżej tabelę, która zawiera opisy bardzo różnorodnego zbioru bohaterów literackich.

Nazwisko	Płeć	Zawód	Planeta macierzysta
Ford Prefect	mężczyzna	badacz	Betelgeza Siedem
Arthur Dent	mężczyzna	pan kanapka	Ziemia
Tricia McMillan	kobieta	matematyczka	Ziemia
Marvin	nieznana	paranoiczny android	nieznana

Przypomnij sobie, jak na początku tego rozdziału tworzyliśmy słownik o nazwie `person3`, który miał przechowywać dane dotyczące osoby o nazwisku Ford Prefect. Struktura ta została pokazana poniżej.

```
person3 = { 'Nazwisko': 'Ford Prefect',
            'Płeć': 'mężczyzna',
            'Zawód': 'badacz',
            'Planeta macierzysta': 'Betelgeza Siedem' }
```

Zamiast powoływać do życia cztery oddzielne zmienne słownikowe dla każdego z wierszy naszej tabeli (i borykać się z nimi potem), utwórzmy pojedynczą zmienną słownikową o nazwie `people`, a następnie wykorzystajmy ją do przechowywania dowolnej liczby innych słowników.

Aby zrealizować ten plan, na początek utwórzmy pusty słownik `people`, a potem przypiszmy dane Forda Prefecta do odpowiedniego klucza, tak jak widać poniżej.

Zacznijmy od nowego, pustego słownika.

```
>>> people = {}
>>> people['Ford'] = { 'Nazwisko': 'Ford Prefect',
                       'Płeć': 'mężczyzna',
                       'Zawód': 'badacz',
                       'Planeta macierzysta': 'Betelgeza Siedem' }
```

Kluczem jest łańcuch znakowy 'Ford', a wartością — kolejny słownik.

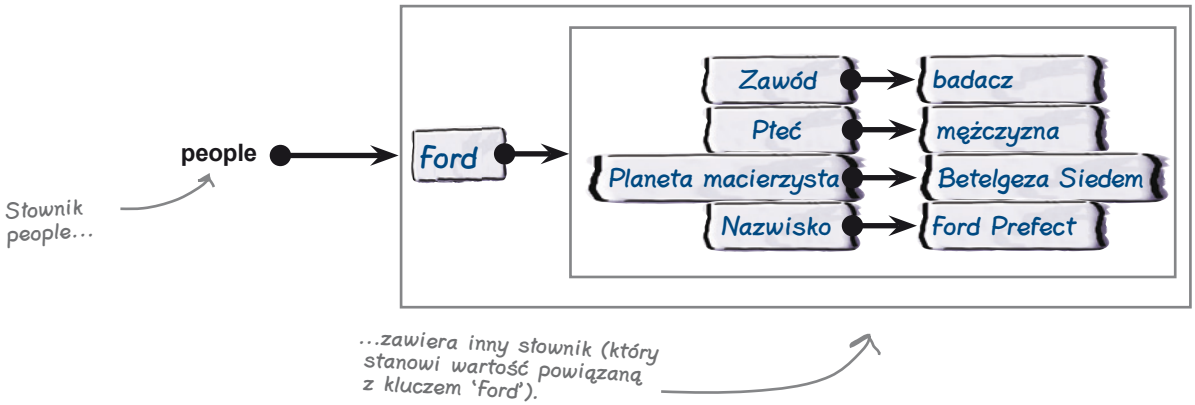
Słownik zawierający słownik

Gdy mamy już nasz słownik `people`, a w nim dodany jeden wiersz danych (opisujący Forda), możemy poprosić interpreter o wyświetlenie zawartości tego słownika w wierszu poleceń powłoki. Otrzymane w ten sposób dane wyjściowe na pierwszy rzut oka będą wyglądały nieco myląco, ale szybko przekonasz się, że wszystko jest na swoim miejscu.

Słownik osadzony w słowniku — zwróć uwagę na dodatkowy nawias klamrowy.

```
>>> people
{'Ford': {'Nazwisko': 'Ford Prefect', 'Płeć': 'mężczyzna',
'Zawód': 'badacz', 'Planeta macierzysta': 'Betelgeza Siedem'}}
```

W słowniku `people` znajduje się tylko jeden (jak dotąd) osadzony słownik, dlatego określanie tej struktury mianem „słownika słowników” jest trochę na wyrost (użycie liczby mnogiej nie jest tu na razie w pełni uzasadnione). Poniżej przedstawiono, jak interpreter „widzi” słownik `people`.



Możemy teraz kontynuować dodawanie danych pochodzących z trzech pozostałych wierszy naszej tabeli.

Dane Arthura

```
>>> people['Arthur'] = { 'Nazwisko': 'Arthur Dent',
                         'Płeć': 'mężczyzna',
                         'Zawód': 'pan kanapka',
                         'Planeta macierzysta': 'Ziemia' }
>>> people['Trillian'] = { 'Nazwisko': 'Tricia McMillan',
                           'Płeć': 'kobieta',
                           'Zawód': 'matematyczka',
                           'Planeta macierzysta': 'Ziemia' }
>>> people['Robot'] = { 'Nazwisko': 'Marvin',
                        'Płeć': 'nieznana',
                        'Zawód': 'paranoiczny android',
                        'Planeta macierzysta': 'nieznana' }
```

Dane Tricii powiązane z kluczem 'Trillian'

Dane Marvina powiązane z kluczem 'Robot'

Słownik słowników (znany również jako tabela)

Mając słownik `people` zapełniony czterema osadzonymi słownikami, możemy poprosić interpreter o wyświetlenie jego zawartości w wierszu poleceń powłoki.

Po zrobieniu tego ujrzymy na ekranie niesamowity mętlik danych (tak jak zostało to pokazane poniżej).

Pomimo całego nieporządku wszystkie nasze dane są na miejscu. Zauważ, że każdy otwierający nawias klamrowy oznacza początek nowego słownika, podczas gdy zamykający nawias klamrowy stanowi jego koniec. Śmiało, policz wszystkie te nawiasy (powinno być pięć otwierających i pięć zamykających).

Trochę trudno się to czyta, ale wszystkie dane znajdują się tam, gdzie trzeba.

```
>>> people
{'Ford': {'Nazwisko': 'Ford Prefect', 'Płeć': 'mężczyzna', 'Zawód':
'badacz', 'Planeta macierzysta': 'Betelgeza Siedem'}, 'Arthur':
{'Nazwisko': 'Arthur Dent', 'Płeć': 'mężczyzna', 'Zawód': 'pan
kanapka', 'Planeta macierzysta': 'Ziemia'}, 'Trillian': {'Nazwisko':
'Tricia McMillan', 'Płeć': 'kobieta', 'Zawód': 'matematyczka',
'Planeta macierzysta': 'Ziemia'}, 'Robot': {'Nazwisko': 'Marvin',
'Płeć': 'nieznana', 'Zawód': 'paranoiczny android', 'Planeta
macierzysta': 'nieznana'}}
```

Interpreter po prostu wyrzucił dane na ekran. Czy istnieje jakaś możliwość pokazania ich w bardziej przyswajalnej postaci?

Tak, możemy sprawić, że dane będzie się dało łatwo czytać.

Moglibyśmy przejść do wiersza poleceń powłoki i po znaku zachęty >>> napisać kod krótkiej pętli `for`, która iterowałaby po każdym kluczu należącym do słownika `people`. Zagnieżdżona pętla `for` mogłaby z kolei przetwarzać każdy osadzony słownik, dzięki czemu zapewnilibyśmy, że dane wyjściowe dałoby się w miarę łatwo czytać z ekranu.

Moglibyśmy... ale nie będziemy tego robić, ponieważ zrobił to już za nas ktoś inny.

Ładne wyświetlanie złożonych struktur danych

Biblioteka standardowa zawiera moduł o nazwie `pprint`, zapewniający możliwość wyświetlania każdej podanej struktury danych w formie, którą łatwo się czyta. Nazwa modułu, `pprint`, stanowi skrót od angielskich słów „pretty print", oznaczających ładny druk lub ładne wyświetlanie.

Skorzystajmy z tego modułu w stosunku do naszego słownika (słowników) `people`. Poniżej raz jeszcze wyświetlamy dane „cięgiem" w wierszu poleceń, a następnie importujemy moduł `pprint` i wywołujemy metodę o tej samej nazwie, aby uzyskać dane wyjściowe w formacie, o który nam chodzi.

Nasz słownik słowników trudno się czyta.

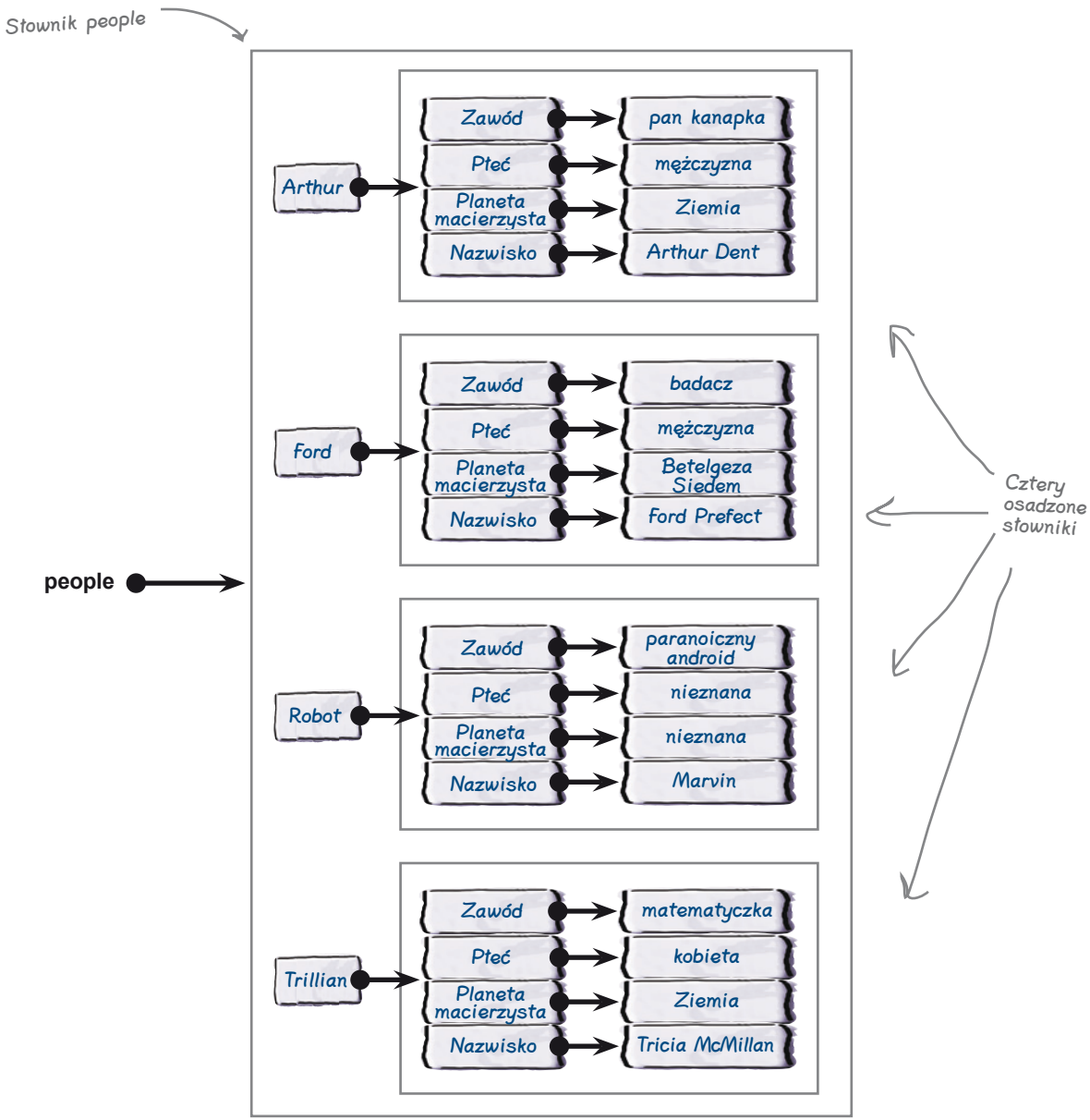
```
>>> people
{'Ford': {'Nazwisko': 'Ford Prefect', 'Płeć': 'mężczyzna', 'Zawód':
'badacz', 'Planeta macierzysta': 'Betelgeza Siedem'}, 'Arthur':
{'Nazwisko': 'Arthur Dent', 'Płeć': 'mężczyzna', 'Zawód': 'pan
kanapka', 'Planeta macierzysta': 'Ziemia'}, 'Trillian': {'Nazwisko':
'Tricia McMillan', 'Płeć': 'kobieta', 'Zawód': 'matematyczka',
'Planeta macierzysta': 'Ziemia'}, 'Robot': {'Nazwisko': 'Marvin',
'Płeć': 'nieznana', 'Zawód': 'paranoiczny android', 'Planeta
macierzysta': 'nieznana'}}
>>>
>>> import pprint
>>>
>>> pprint.pprint(people)
{'Arthur': {'Nazwisko': 'Arthur Dent',
            'Planeta macierzysta': 'Ziemia',
            'Płeć': 'mężczyzna',
            'Zawód': 'pan kanapka'},
 'Ford': {'Nazwisko': 'Ford Prefect',
          'Planeta macierzysta': 'Betelgeza Siedem',
          'Płeć': 'mężczyzna',
          'Zawód': 'badacz'},
 'Robot': {'Nazwisko': 'Marvin',
           'Planeta macierzysta': 'nieznana',
           'Płeć': 'nieznana',
           'Zawód': 'paranoiczny android'},
 'Trillian': {'Nazwisko': 'Tricia McMillan',
              'Planeta macierzysta': 'Ziemia',
              'Płeć': 'kobieta',
              'Zawód': 'matematyczka'}}
```

Importujemy moduł pprint, a następnie wywołujemy metodę pprint, aby załatwić sprawę.

Te dane wyjściowe o wiele łatwiej się czyta. Zauważ, że nadal mamy tu pięć otwierających i pięć zamykających nawiasów klamrowych. Jedyna różnica polega na tym, że dzięki funkcji pprint o wiele łatwiej je teraz zobaczyć (i policzyć).

Wizualizacja złożonych struktur danych

Zaktualizujmy nasz diagram przedstawiający to, jak interpreter „widzi” słownik słowników `people`, który został wypełniony danymi.



Na tym etapie może pojawić się zupełnie uzasadnione pytanie, które brzmi: *Teraz, gdy mamy już wszystkie te dane zapisane w słowniku słowników, to jak możemy się do nich dostać?* Odpowiemy na to pytanie na następnej stronie.

Uzyskiwanie dostępu do złożonych struktur danych

Naszą tabelę danych mamy już zapisaną pod postacią słownika `people`. Przypomnijmy sobie teraz jej oryginalną formę, od której zaczynaliśmy.

Nazwisko	Płeć	Zawód	Planeta macierzysta
Ford Prefect	mężczyzna	badacz	Betelgeza Siedem
Arthur Dent	mężczyzna	pan kanapka	Ziemia
Tricia McMillan	kobieta	matematyczka	Ziemia
Marvin	nieznana	paranoiczny android	nieznana

Gdybyśmy zostali poproszeni o sprawdzenie, z czego utrzymuje się Arthur, zaczęlibyśmy od przejrzenia kolumny **Nazwisko** w poszukiwaniu jego nazwiska, a następnie przesunęlibyśmy wzrok wzdłuż odpowiedniego wiersza danych aż do miejsca, w którym znaleźlibyśmy kolumnę **Zawód**, gdzie moglibyśmy odczytać, że jest on „panem kanapką".

Gdy zachodzi potrzeba uzyskania dostępu do danych znajdujących się w złożonej strukturze (takiej jak słownik słowników `people`), procedura powinna wyglądać całkiem podobnie. Sposób postępowania przedstawimy Ci poniżej przy użyciu wiersza poleceń powłoki Pythona.

Zaczynamy od odszukania danych opisujących Arthura w słowniku `people`, co można zrobić, umieszczając odpowiadający mu klucz w nawiasach kwadratowych.

Poproś o dane z wiersza Arthura.

```
>>> people['Arthur']
{'Nazwisko': 'Arthur Dent', 'Płeć': 'mężczyzna', 'Zawód': 'pan
kanapka', 'Planeta macierzysta': 'Ziemia'}
```

Wiersz danych słownika powiązanych z kluczem 'Arthur'

Jeśli udało nam się już odnaleźć wiersz danych opisujących Arthura, możemy poprosić o wartość powiązaną z kluczem `Zawód`. W tym celu skorzystamy z **drugiej** pary nawiasów kwadratowych, aby za pomocą odpowiedniego „indeksu" odwołać się do powiązanych z nim danych w słowniku Arthura.

Określ wiersz.

Określ kolumnę.

```
>>> people['Arthur']['Zawód']
'pan kanapka'
```

Zastosowanie podwójnych nawiasów kwadratowych umożliwia Ci uzyskanie dostępu do dowolnej wartości zapisanej w tablicy poprzez określenie wiersza i kolumny, które Cię interesują. Wiersz odpowiada kluczowi wykorzystywanemu w słowniku obejmującym (w naszym przykładzie jest to słownik `people`), kolumna zaś odpowiada dowolnemu z kluczy używanych w słowniku osadzonym.

Dane są tak skomplikowane, jakimi je uczynisz

Niezależnie od tego, czy masz do czynienia z niewielką ilością danych (na przykład tworzących prostą listę), czy też z czymś znacznie bardziej złożonym (takim jak słownik słowników), dobrze jest wiedzieć, że cztery wbudowane struktury danych Pythona są w stanie spełnić Twoje potrzeby związane z przechowywaniem informacji. A tym, co okazuje się tu szczególnie miłe, jest dynamiczna natura tworzonych przez Ciebie struktur danych, przejawiająca się w tym, że każda struktura — oprócz krotki — może powiększać się i kurczyć w zależności od potrzeb, tak że nie musisz zaprzątać sobie głowy wewnętrznymi szczegółami związanymi z wszelkimi niezbędnymi operacjami rezerwowania i zwalniania pamięci, ponieważ o wszystko dba interpreter Pythona.

Nie zakończyliśmy jeszcze omawiania możliwości Pythona związanych z danymi, dlatego powrócimy do tego tematu ponownie w dalszej części tej książki. Na tym etapie dysponujesz już jednak wystarczającą wiedzą, aby zająć się kolejnymi tematami.

W następnym rozdziale zaczniemy omawiać techniki umożliwiające efektywne ponowne wykorzystywanie kodu w języku Python, prezentując jeden z podstawowych sposobów zapewniających tę możliwość, jakim jest stosowanie funkcji.

Kod przykładów z rozdziału 3. (1. z 2)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
   if letter in vowels:
      found[letter] += 1

for k, v in sorted(found.items()):
   print(k, 'znaleziono', v, 'raz(y).')
```

To jest kod programu vowels4.py, którego zadaniem było zliczanie samogłosek w podanym słowie. Kod ten powstał (w pewnej mierze) na bazie programu vowels3.py, który został po raz pierwszy zaprezentowany w rozdziale 2.

Próbując pozbyć się kodu inicjalizującego słownik, utworzyliśmy program vowels5.py, który uległ awarii z powodu wystąpienia błędu czasu wykonania (związanego z brakiem inicjalizacji liczników wystąpień poszczególnych samogłosek).

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")

found = {}

for letter in word:
   if letter in vowels:
      found[letter] += 1

for k, v in sorted(found.items()):
   print(k, 'znaleziono', v, 'raz(y).')
```

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")

found = {}

for letter in word:
   if letter in vowels:
      found.setdefault(letter, 0)
      found[letter] += 1

for k, v in sorted(found.items()):
   print(k, 'znaleziono', v, 'raz(y).')
```

W programie vowels6.py udało nam się wyeliminować błąd czasu wykonania dzięki zastosowaniu metody setdefault, która jest dostępna w przypadku każdego słownika (jej działanie polega na przypisaniu standardowej wartości do danej powiązanej z kluczem, jeśli dana ta nie została jeszcze wcześniej ustawiona).

Kod przykładów z rozdziału 3. (2. z 2)

```
vowels = set('aeiou')
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Ostateczna wersja programu vowels7.py, w której wykorzystaliśmy możliwości oferowane przez kolejną strukturę danych Pythona, czyli zbiór, aby znacznie skrócić kod programu vowels3.py korzystający z list, zapewniając przy tym dokładnie ten sam efekt działania.

Czy nie zabrakło tu przypadkiem przykładowego programu, w którym skorzystalibyśmy z możliwości oferowanych przez krotki?

Zgadza się, zabrakło. Ale to nie problem.

W tym rozdziale nie zaprezentowaliśmy krotek za pomocą przykładowego programu, ponieważ pełnych możliwości tych struktur danych nie da się sensownie przedstawić w oderwaniu od funkcji. Jak wcześniej stwierdziliśmy, już wkrótce wrócimy do tematu krotek. Zajmiemy się nimi ponownie, gdy będziemy omawiać funkcje (czyli w następnym rozdziale), jak również w kilku kolejnych miejscach w dalszej części książki. Za każdym razem, gdy krotki pojawią się w jej treści, na pewno wskażemy Ci ich zastosowanie. Kontynuując swoją podróż z Pythonem, przekonasz się, że krotki występują dosłownie wszędzie.

Funkcje i moduły

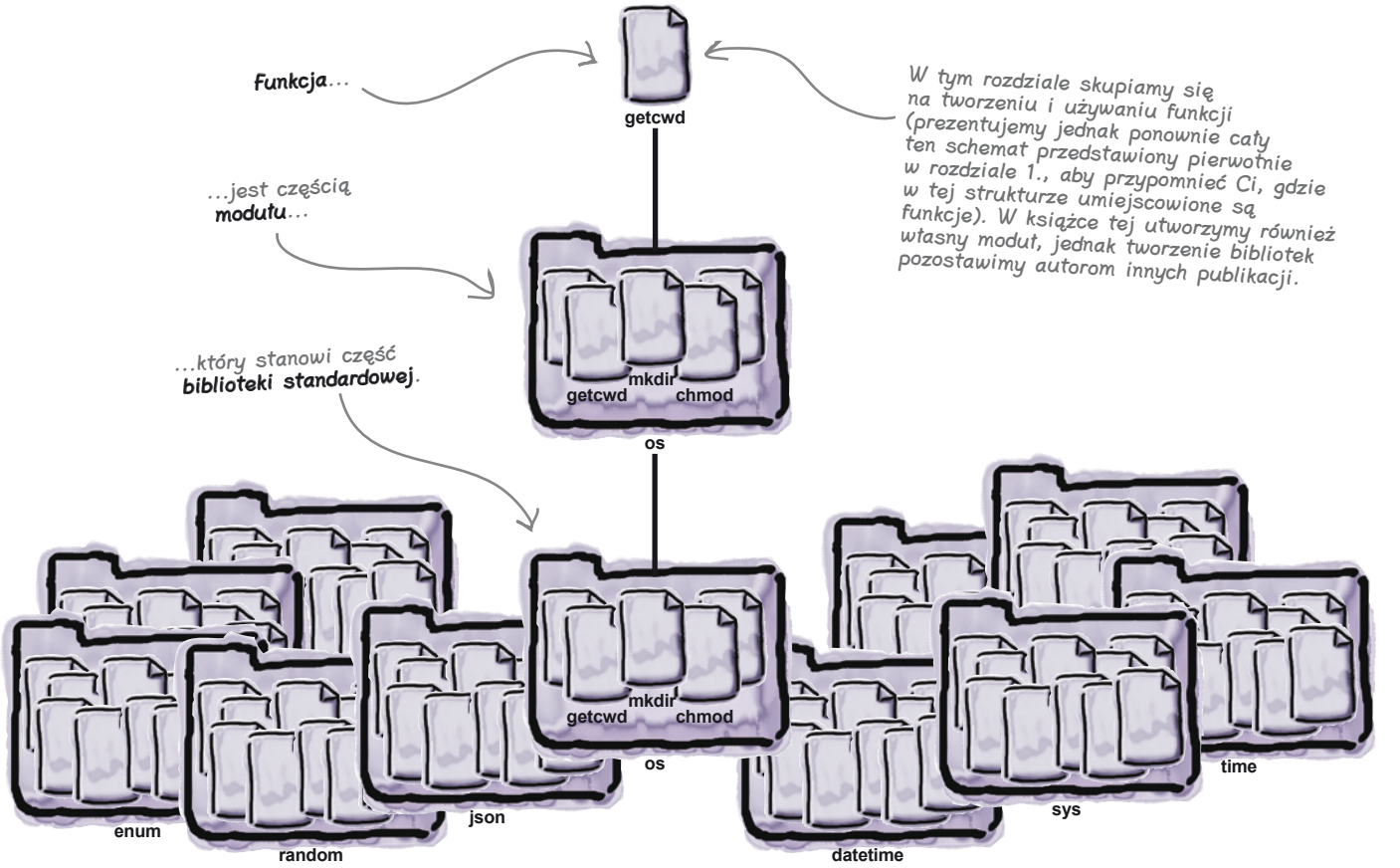
Ponowne wykorzystanie kodu ma kluczowe znaczenie dla budowy prostego w utrzymaniu systemu.

A jeśli chodzi o wielokrotne używanie kodu w Pythonie, wszystko zaczyna się od skromnej **funkcji** i na niej też kończy. Weź kilka wierszy kodu, nadaj im nazwę i masz funkcję (która może być wielokrotnie wykorzystywana). Weź kolekcję funkcji i spakuj je do jednego pliku, a dostaniesz **moduł** (którego również można używać wiele razy). Prawdą jest, że *dobrze się dzielić*. Zanim skończysz czytać ten rozdział, będziesz na dobrej drodze do **udostępniania** i **ponownego wykorzystywania** swojego kodu, a to dzięki poznaniu sposobu działania funkcji i modułów w języku Python.

Ponowne wykorzystywanie kodu za pomocą funkcji

Choć kilka wierszy kodu może w Pythonie zdziałać bardzo dużo, wcześniej czy później przekonasz się, że Twoja baza kodu rośnie, a gdy tak się dzieje, sprawy szybko zaczynają wymykać się spod kontroli. Program, który na początku miał zaledwie 20 wierszy kodu Pythona, w jakiś magiczny sposób wypączkował, aby osiągnąć 500 lub nawet więcej wierszy! Gdy tak się dzieje, czas rozważyć strategie, które możesz zastosować, aby zmniejszyć złożoność Twojej bazy kodu.

Podobnie jak ma to miejsce w innych językach programowania, Python wspiera **modułowość**, dzięki której możesz podzielić duże partie kodu na mniejsze, łatwiejsze w utrzymaniu kawałki. Robi się to, tworząc **funkcje**, które możesz traktować jak nazwane fragmenty kodu. Przypomnij sobie przedstawiony w rozdziale 1. schemat przedstawiający związki występujące pomiędzy funkcjami, modułami i biblioteką standardową.



W rozdziale tym mamy zamiar skupić się na działaniach związanych z tworzeniem własnych funkcji, takich jak ta, która jest widoczna w górnej części powyższego rysunku. Gdy będziesz już wiedział, jak to zrobić, pokażemy Ci, jak tworzy się moduł.

Wprowadzenie do funkcji

Zanim zaczniemy przerabiać nasz istniejący kod na funkcję, poświęćmy trochę czasu na analizę anatomii *dowolnej* funkcji w Pythonie. Gdy wprowadzenie to dobiegnie końca, przyjrzymy się naszemu istniejącemu kodowi i przejdziemy kroki konieczne, aby nadać mu formę funkcji, której będziesz mógł wielokrotnie używać.

Nie przejmuj się teraz szczegółami. Chodzi nam tu jedynie o to, abyś nabrał ogólnego pojęcie o tym, jak wyglądają funkcje w Pythonie. I tym właśnie zajmiemy się na tej i kolejnej stronie. Po szczegóły, które powinieneś poznać, będziemy sukcesywnie sięgać wraz z tym, jak rozdział ten będzie się rozwijał. Przedstawione poniżej okno edycji IDLE zawiera szablon, z którego możesz korzystać, tworząc swoje funkcje. Przyglądając mu się, rozważ następujące rzeczy:

1. **Funkcje wprowadzają dwa nowe słowa kluczowe: `def` oraz `return`.**
   Obydwa te słowa są w edytorze IDLE wyświetlane na pomarańczowo. Słowo `def` poprzedza nazwę funkcji (wyświetlaną na niebiesko) oraz wszystkie szczegóły na temat argumentów, które może ona przyjmować. Użycie słowa `return` jest opcjonalne; jest ono wykorzystywane do przekazania wartości do kodu, z którego wywołana została funkcja.

2. **Funkcje mogą przyjmować argumenty (dane).**
   Funkcja może przyjmować wartości argumentów (stanowiące dane wejściowe funkcji). Listę parametrów podaje się w nawiasie po nazwie funkcji, w tym samym wierszu, w którym znajduje się słowo kluczowe `def`.

3. **Funkcje zawierają kod oraz (zwykle) dokumentację.**
   Kod tworzący funkcję jest wcięty o jeden poziom bardziej niż wiersz ze słowem `def` i powinien zawierać komentarze wszędzie tam, gdzie ma to sens. Poniżej zademonstrowaliśmy dwa sposoby dodawania komentarzy: przy użyciu łańcucha znakowego ujętego w trzy znaki cudzysłowu na początku i końcu (wyświetlanego w oknie na zielono i znanego pod nazwą **tekstu dokumentującego**) oraz w postaci komentarzy jednowierszowych, zaczynających się od symbolu # (które wyświetlane są na czerwono).

W wierszu zaczynającym się od słowa „def" określa się nazwę i listę argumentów funkcji.

Tekst dokumentujący opisuje zastosowanie funkcji.

function_template.py - C:/Users/lsuma/Desktop/Python/...

File Edit Format Run Options Window Help

```
def opisowa_nazwa(opcjonalne_argumenty):
    """Tekst dokumentujący."""
    # Kod Twojej funkcji znajduje się tutaj.
    # Kod Twojej funkcji znajduje się tutaj.
    # Kod Twojej funkcji znajduje się tutaj.
    return opcjonalna_wartość
```

Ln: 8 Col: 0

To jest wygodny szablon funkcji.

Twój kod powinien znaleźć się tutaj (w miejscu zajmowanym teraz przez te komentarze jednowierszowe).

Porady maniaka

Nazwa „funkcja" używana jest w Pythonie do opisu fragmentu kodu, który może być wielokrotnie wykorzystywany. W innych językach na określenie tego rodzaju konstrukcji używane są słowa „procedura", „podprogram" i „metoda". Jeśli funkcja należy do klasy Pythona, znana jest jako „metoda". O klasach i metodach w języku Python dowiesz się z jednego z kolejnych rozdziałów tej książki.

Co z informacją na temat typu?

Przyjrzyj się naszemu szablonowi funkcji raz jeszcze. Czy oprócz kodu, który ma być wykonany w jej ramach, brakuje tu jeszcze czegoś? Czy nie pominięto w nim jakiejś rzeczy, którą spodziewałeś się zobaczyć? Przyjrzyj się uważnie.

function_template.py - C:/Users/lsuma/Desktop/Python/...

File Edit Format Run Options Window Help

```
def opisowa_nazwa(opcjonalne_argumenty):
    """Tekst dokumentujący."""
    # Kod Twojej funkcji znajduje się tutaj.
    # Kod Twojej funkcji znajduje się tutaj.
    # Kod Twojej funkcji znajduje się tutaj.
    return opcjonalna_wartość
```

Ln: 8 Col: 0

Czy w tym szablonie funkcji czegoś nie brakuje?

Trochę osłupiałem na widok tego szablonu funkcji. Skąd interpreter będzie wiedział, jakich typów argumentów ma się spodziewać i jaki będzie typ wartości zwracanej?

Nie będzie wiedział, ale Ty nie musisz się tym martwić.

Interpreter języka Python nie wymusza na Tobie określania typów argumentów funkcji ani zwracanej przez nią wartości. Niewykluczone, że dość mocno Cię to zaskoczyło, jeśli korzystałeś wcześniej z innych języków programowania, w których było to konieczne. Nie powinieneś się tym jednak martwić.

Python umożliwia Ci przekazywanie dowolnego *obiektu* w roli argumentu funkcji, a także przekazywanie z powrotem dowolnego *obiektu* jako wartości zwracanej. Interpreter nie dba o typ tych obiektów ani go nie sprawdza (troszczy się tylko o to, aby obiekty te zostały podane).

W języku Python 3 można *wskazać* spodziewane typy argumentów i wartości zwracanych; pokażemy to zresztą w dalszej części niniejszego rozdziału. Skorzystanie z tej możliwości nie skutkuje jednak magicznym włączeniem mechanizmu kontroli typów, ponieważ Python *nigdy* nie sprawdza typów argumentów ani wartości zwracanych.

Nazywanie fragmentu kodu przy użyciu słowa kluczowego def

Jeśli udało Ci się już zidentyfikować kawałek swojego kodu Pythona, z którego chciałbyś ponownie skorzystać, nadszedł czas na utworzenie funkcji. Robi się to za pomocą słowa kluczowego `def` (które stanowi skrót od *definiowania*). Po nim powinny się pojawić: nazwa funkcji, opcjonalnie pusta lista argumentów (umieszczona w nawiasie okrągłym), dwukropek, a następnie co najmniej jeden wiersz wciętego kodu stanowiący ciało funkcji (może ich być oczywiście więcej).

Przypomnij sobie program *vowels7.py*, przedstawiony na końcu poprzedniego rozdziału. Jego zadaniem było wyświetlenie na ekranie samogłosek znalezionych w podanym przez użytkownika słowie, tak jak zostało to pokazane poniżej.

Weź zbiór samogłosek...

...oraz słowo...

...a następnie znajdź część wspólną.

```
vowels = set('aeiou')
word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Wyświetl ewentualne wyniki.

To jest kod programu vowels7.py, który został przedstawiony na końcu rozdziału 3.

Wyobraźmy sobie, że planujesz używać tych pięciu wierszy kodu wielokrotnie w znacznie większym programie. Kopiowanie i wklejanie tego fragmentu wszędzie tam, gdzie jest on potrzebny, jest ostatnią rzeczą, którą chciałbyś zrobić, dlatego aby ułatwić sobie panowanie nad kodem i zapewnić, że będziesz musiał opiekować się tylko **jedną kopią** tego wycinka, postanowiłeś utworzyć z niego funkcję.

Pokażemy Ci, jak to zrobić, korzystając (na tym etapie) z powłoki Pythona. Aby przekształcić te pięć wierszy kodu w funkcję, skorzystamy ze słowa kluczowego `def`, które wskazuje, że w tym miejscu zaczyna się jej definicja. Następnie nadamy funkcji opisową nazwę (jest to *zawsze* dobry pomysł), w nawiasie okrągłym podamy opcjonalnie pustą listę argumentów, po której wstawimy znak dwukropka. Ostatnią czynnością będzie dodanie odpowiednich wcięć (w odniesieniu do słowa kluczowego `def`).

Poświęć nieco czasu, aby wybrać dla funkcji nazwę dobrze opisującą jej działanie.

Zacznij od słowa kluczowego def.

Nadaj swojej funkcji dobrą, opisową nazwę.

Podaj opcjonalną listę argumentów w nawiasie; w tym przypadku funkcja nie przyjmuje żadnych argumentów, więc lista jest pusta.

Nie zapomnij o znaku dwukropka.

```
>>> def search4vowels():
    vowels = set('aeiou')
    word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Pięć wierszy kodu tworzących program vowels7.py zostało odpowiednio wciętych.

Jako że korzystamy tu z powłoki, pamiętaj o DWUKROTNYM naciśnięciu klawisza Enter, aby potwierdzić, że ten wcięty fragment kodu został zakończony.

Teraz, gdy nasza funkcja jest już zdefiniowana, wywołajmy ją, aby się przekonać, czy działa zgodnie z oczekiwaniami.

Wywoływanie Twojej funkcji

Aby wywołać funkcję w Pythonie, należy podać jej nazwę oraz wartości wszystkich argumentów, które spodziewa się ona otrzymać. Jako że funkcja `search4vowels` nie przyjmuje (obecnie) żadnych argumentów, możemy ją wywołać z pustą listą argumentów, co zostało pokazane poniżej.

```
>>> search4vowels()
Podaj słowo, w którym należy wyszukać samogłoski: mierni
e
i
```

Ponowne wywołanie funkcji powoduje, że zostaje ona wykonana po raz kolejny.

```
>>> search4vowels()
Podaj słowo, w którym należy wyszukać samogłoski: galaktyka
a
```

Nie ma tu żadnych niespodzianek: wywołanie funkcji powoduje wykonanie jej kodu.

Edytuj kod swoich funkcji w oknie edytora, nie w oknie powłoki

Kod funkcji `search4vowels` został wprowadzony po znaku zachęty powłoki >>> i wygląda tak, jak widać poniżej.

Kod naszej funkcji w formie, w jakiej został wpisany w wierszu poleceń powłoki.

```
>>> def search4vowels():
    vowels = set('aeiou')
    word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Aby dalej rozwijać ten kod, możesz przywoływać go w wierszu poleceń i tam dokonywać odpowiednich zmian, jednak sposób ten szybko okaże się niepraktyczny. Przypomnij sobie, że gdy pracując z kodem w powłoce, zaobserwujesz, że jego długość przekracza kilka wierszy, powinieneś skopiować go do okna edycji środowiska IDLE. W nim będzie Ci o wiele wygodniej modyfikować kod, dlatego zanim ruszymy dalej, przenieśmy go do okna edycji.

Upewnij się, że zapisałeś swój kod w pliku vsearch.py po skopiowaniu do niego kodu funkcji z okna powłoki.

Otwórz nowe, puste okno edycji środowiska IDLE, skopiuj kod funkcji z okna powłoki (nie uwzględniając jednak znaku zachęty >>>), a następnie wklej go do okna edycji. Gdy upewnisz się, że formatowania i wcięcia kodu są poprawne, przed przejściem do kolejnych działań zapisz swój program w pliku *vsearch.py*.

Wprowadzając zmiany, korzystaj z edytora IDLE

Poniżej możesz zobaczyć, jak kod programu *vsearch.py* prezentuje się w oknie edycji środowiska IDLE.

Kod funkcji znajduje się teraz w oknie edycji środowiska IDLE i został zapisany w pliku o nazwie vsearch.py.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels():
    vowels = set('aeiou')
    word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 8 Col: 0

Gdy naciśniesz klawisz *F5* w oknie edycji, zdarzą się dwie rzeczy: powłoka IDLE zostanie przywołana na wierzch oraz zrestartowana. Nic jednak nie ukaże się na ekranie. Spróbuj to teraz zrobić, aby przekonać się, co mamy na myśli: naciśnij klawisz *F5*. Powodem, dla którego na ekranie nie widać żadnego komunikatu, jest to, że nie wywołaliśmy jeszcze naszej funkcji. Zaraz to zrobimy, najpierw jednak wprowadźmy do funkcji jedną zmianę. Będzie to zmiana niewielka, lecz istotna.

Nie panikuj, jeśli środowisko IDLE wyświetli komunikat błędu, gdy naciśniesz klawisz F5! Wróć do okna edycji i sprawdź, czy Twój kod jest dokładnie taki sam jak nasz, a następnie spróbuj ponownie.

Dodajmy trochę dokumentacji w górnej części kodu funkcji.

Aby do dowolnego kodu dodać wielowierszowy komentarz (**tekst dokumentujący**), umieść go pomiędzy dwoma sekwencjami, z których każda składa się z trzech znaków cudzysłowu.

Poniżej raz jeszcze znajdziesz kod programu *vsearch.py*, w którym tym razem dodany został tekst dokumentujący funkcję. Wprowadź analogiczną zmianę w swojej kopii kodu.

Do kodu funkcji został dodany tekst dokumentujący, który (w skrócie) opisuje jej przeznaczenie.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels():
    """Wyświetla samogłoski znalezione w słowie podanym przez użytkownika."""
    vowels = set('aeiou')
    word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 9 Col: 0

O co chodzi z tymi wszystkimi łańcuchami znakowymi?

Spójrz raz jeszcze na naszą funkcję w obecnej postaci. Zwróć szczególną uwagę na trzy łańcuchy znakowe znajdujące się w jej kodzie, które zostały przez środowisko IDLE wyróżnione kolorem zielonym.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels():
    """Wyświetla samogłoski znalezione w słowie podanym przez użytkownika."""
    vowels = set('aeiou')
    word = input("Podaj słowo, w którym należy wyszukać samogłoski: ")
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 9 Col: 0

Zapewniana przez środowisko IDLE funkcja kolorowania składni sygnalizuje, że mamy tu pewien problem ze spójnością użycia cudzysłowów. Kiedy powinniśmy stosować poszczególne style?

Znaczenie cudzysłowów ograniczających łańcuchy znakowe

W języku Python łańcuchy znakowe mogą być umieszczane w pojedynczych cudzysłowach (`'`), w podwójnych cudzysłowach (`"`) lub pomiędzy trójkami cudzysłowów (`"""` lub `'''`).

Jak już wcześniej wspominaliśmy, łańcuch znakowy ograniczony z obydwu stron trzema znakami cudzysłowu jest określany mianem **tekstu dokumentującego** (ang. *docstring*), ponieważ ciągi tego rodzaju są używane głównie do dokumentowania przeznaczenia funkcji (tak jak zostało to przedstawione powyżej). Teksty takie możesz umieszczać pomiędzy znakami `"""` lub `'''`, lecz większość programistów Pythona woli korzystać z tych pierwszych. Teksty dokumentujące mają tę ciekawą właściwość, że mogą się rozciągać na wiele wierszy (w innych językach programowania konstrukcja ta znana jest nieraz pod nazwą „heredoc”).

Łańcuchy znakowe ograniczone znakami cudzysłowu pojedynczego (`'`) lub podwójnego (`"`) **nie mogą** rozciągać się na wiele wierszy, co oznacza, że musisz zakończyć tego typu łańcuch za pomocą odpowiedniego znaku jeszcze w tej samej linii (jest tak, ponieważ Python wykorzystuje koniec wiersza w roli znaku kończącego instrukcję).

To od Ciebie zależy, z jakich znaków będziesz korzystał do ograniczania swoich łańcuchów znakowych, powinieneś jednak wiedzieć, że użycie w tym celu pojedynczego cudzysłowu jest bardzo popularnym rozwiązaniem stosowanym przez większość programistów Pythona. Na cokolwiek jednak byś się tu zdecydował, powinieneś być w tym konsekwentny.

Staraj się zachować konsekwencję w stosowaniu cudzysłowów. Jeśli to możliwe, korzystaj ze znaków pojedynczego cudzysłowu.

Stosuj najlepsze wzorce zebrane w dokumentach PEP

Jeśli chodzi o formatowanie kodu (nie tylko występujących w nim łańcuchów znakowych), społeczność programistów Pythona poświęciła sporo czasu na opracowanie i udokumentowanie najlepszego wzorca, którym należy się tu kierować. Ten wzorzec znany jest jako **PEP 8**. PEP to skrót od angielskich słów *Python Enhancement Proposal*, które po polsku oznaczają mniej więcej propozycję rozszerzenia Pythona.

Istnieje bardzo wiele dokumentów PEP, a ich głównym celem jest doprecyzowanie proponowanych i zaimplementowanych rozszerzeń języka Python, mogą one jednak też dokumentować pewne wskazówki (dotyczące tego, co należy lub czego nie należy robić), jak również opisywać rozmaite procesy związane z tym językiem. Szczegóły dokumentów PEP mogą mieć bardzo techniczny charakter i są (często) dość tajemnicze. Z tego powodu choć ogromna większość programistów Pythona ma świadomość ich istnienia, to bardzo rzadko ma z nimi bezpośredni kontakt. Twierdzenie to jest prawdą w przypadku większości dokumentów PEP, *wyjątkiem* jest tu jednak PEP 8.

Listę dokumentów PEP możesz znaleźć pod adresem *https://www.python.org/dev/peps/*.

PEP 8 to przewodnik stylu dla kodu Pythona. Przeczytanie go zaleca się wszystkim programistom korzystającym z tego języka. Jest to ten dokument, który sugeruje zachowanie spójności w stosowaniu cudzysłowów ograniczających łańcuchy znakowe, o czym była mowa na poprzedniej stronie. Znajdź czas, aby przeczytać PEP 8 chociaż raz. Inny dokument, PEP 257, przedstawia propozycje konwencji określających, jak należy formatować teksty dokumentujące kod. On również wart jest przeczytania.

Poniżej raz jeszcze znajdziesz kod funkcji `search4vowels`, tym razem jednak w formie bardziej zgodnej ze standardami wyznaczanymi przez dokumenty PEP 8 i PEP 257. Wprowadzone zmiany nie są zbyt duże, ale dostosowanie do standardu znaków pojedynczego cudzysłowu wokół łańcuchów znakowych (z wyjątkiem tekstu dokumentującego) sprawia, że kod wygląda nieco lepiej.

To tekst dokumentujący, zgodny z wytycznymi zawartymi w dokumencie PEP 257.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels():
    """Wyświetla samogłoski znalezione w słowie podanym przez użytkownika."""
    vowels = set('aeiou')
    word = input('Podaj słowo, w którym należy wyszukać samogłoski: ')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 9 Col: 0

Podążyliśmy za wskazówką zamieszczoną w dokumencie PEP 8, która dotyczy zachowania spójności w kwestii znaku pojedynczego cudzysłowu stosowanego do ograniczania łańcuchów znakowych.

Nie musisz oczywiście pisać kodu, który *dokładnie* spełnia standardy opisane w dokumencie PEP 8. Wymogów przewodnika stylu nie spełnia na przykład nazwa naszej funkcji, czyli `search4vowels`, ponieważ sugeruje on, aby poszczególne słowa tworzące nazwę funkcji były od siebie oddzielone znakami podkreślenia. Bardziej właściwa według PEP nazwa brzmiałaby więc `search_for_vowels`. Zauważ, że PEP to zbiór wskazówek, a nie reguł. Nie masz obowiązku się z nimi zgadzać, powinieneś je tylko wziąć pod uwagę, a nam podoba się nazwa `search4vowels`.
Mimo wszystko ogromna większość programistów Pythona będzie Ci wdzięczna za pisanie kodu zgodnego ze wskazówkami zawartymi w dokumencie PEP 8, ponieważ kod taki przeważnie łatwiej się czyta niż kod, którego twórcy nie kierowali się tymi radami.

Powróćmy teraz do rozszerzania funkcji `search4vowels` w taki sposób, aby przyjmowała argumenty.

Funkcje przyjmują argumenty

Zamiast pozwalać funkcji `search4vowels` prosić użytkownika o podanie słowa do przeszukania, lepiej byłoby przekazywać jej odpowiednie słowo jako argument wywołania. Wprowadźmy więc odpowiednią zmianę.

Pamiętaj, że „zestaw" to używana w Pythonie nazwa „bloku".

Dodanie argumentu jest bardzo prostą sprawą. Wystarczy wpisać jego nazwę w nawiasie widocznym w wierszu rozpoczynającym się słowem `def`. Ta nazwa argumentu zacznie następnie odgrywać rolę zmiennej w zestawie funkcji. To łatwy krok.

Usuńmy również wiersz kodu proszący użytkownika o podanie słowa, które ma być przeszukiwane. To również prosta sprawa.

Przypomnijmy sobie, jak wygląda w tej chwili nasz kod.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels():
    """Wyświetla samogłoski znalezione w słowie podanym przez użytkownika."""
    vowels = set('aeiou')
    word = input('Podaj słowo, w którym należy wyszukać samogłoski: ')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 9 Col: 0

Tutaj jest widoczna nasza oryginalna funkcja.

Ten wiersz nie jest już do niczego potrzebny.

Po wprowadzeniu dwóch zmian, o których była mowa powyżej, nasza funkcja wygląda już nieco inaczej w oknie edycji środowiska IDLE, o czym możesz się przekonać poniżej. (Zauważ, że zaktualizowaliśmy też tekst dokumentujący, co jest *zawsze* wskazane w takich sytuacjach).

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels(word):
    """Wyświetla samogłoski znalezione w słowie podanym jako argument."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 8 Col: 0

Umieść nazwę argumentu w nawiasie.

Pozbyliśmy się stąd wywołania funkcji input (ponieważ ten wiersz kodu nie jest nam już do niczego potrzebny).

Pamiętaj, aby zapisywać swój kod po każdej dokonanej zmianie, zanim jeszcze naciśniesz klawisz *F5* w celu wypróbowania działania programu.

Jazda próbna

Jeśli Twój kod jest już widoczny w oknie edycji środowiska IDLE (i zapisany w pliku), naciśnij klawisz *F5*, a następnie wywołaj swoją funkcję kilka razy i zobacz, jaki przynosi to efekt.

Aktualny kod funkcji search4vowels

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels(word):
    """Wyświetla samogłoski znalezione w słowie podanym jako argument."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
```

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
========== RESTART: C:/Users/lsuma/Desktop/Python/R04/vsearch.py ==========
>>> search4vowels()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    search4vowels()
TypeError: search4vowels() missing 1 required positional argument: 'word'
>>> search4vowels('mierni')
e
i
>>> search4vowels('mierni', 'galaktyka')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    search4vowels('mierni', 'galaktyka')
TypeError: search4vowels() takes 1 positional argument but 2 were given
>>> |
```

Ln: 18 Col: 4

Choć wywołaliśmy tu funkcję search4vowels aż trzykrotnie, jedynym wywołaniem, które się powiodło, było wywołanie, w którym został przekazany jeden argument będący łańcuchem znakowym. Pozostałe dwa zakończyły się porażką. Poświęć chwilę i przeczytaj zwrócone przez interpreter komunikaty błędów, aby dowiedzieć się, dlaczego poszczególne wywołania się nie powiodły.

Nie istnieją głupie pytania

P: Czy tworząc funkcję z Pythonie, mogę używać wyłącznie jednego argumentu?

O: Nie, możesz korzystać z dowolnej liczby argumentów, która powinna być uzależniona od możliwości zapewnianych przez Twoją funkcję. Celowo zaczynamy tu od bardzo prostego przykładu, ponieważ zamierzamy przejść do bardziej zaawansowanych wraz z rozwojem niniejszego rozdziału. Z argumentami funkcji da się zrobić w Pythonie wiele ciekawych rzeczy. Większość tych możliwości planujemy dokładnie omówić na następnych kilkunastu stronach.

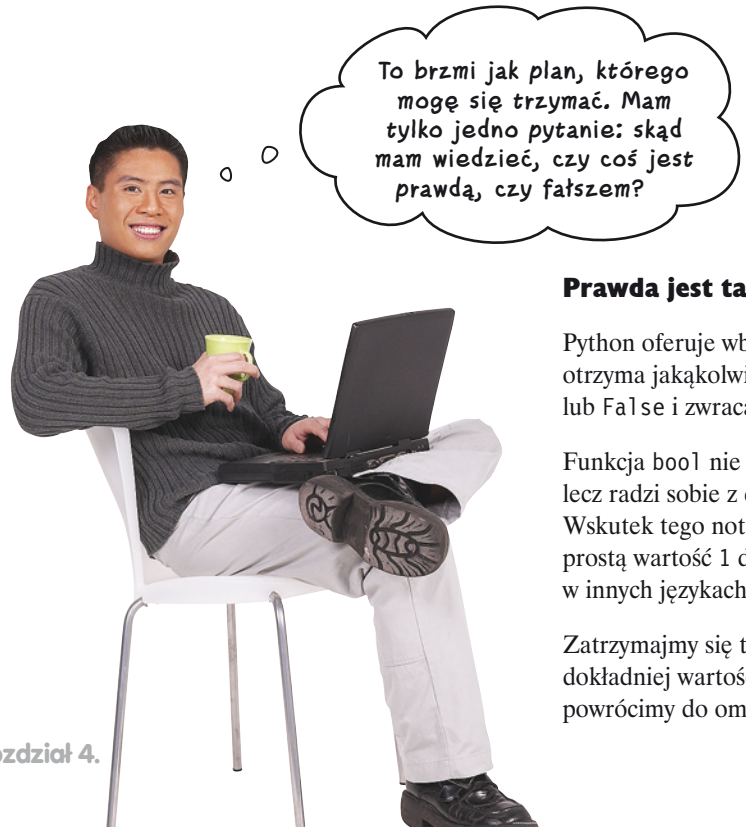
Funkcje zwracają wynik

Wyodrębnianie i nadawanie nazw określonym fragmentom kodu to niejedyne powody, dla których programiści tworzą funkcje i ich używają. Zwykle chcą również, aby ich funkcja zwracała pewną obliczoną wartość, z której może następnie korzystać kod wywołujący funkcję. Aby zapewnić możliwość zwracania wartości (jednej lub wielu), język Python oferuje instrukcję `return`.

Gdy interpreter napotyka instrukcję `return` w zestawie Twojej funkcji, dzieją się dwie rzeczy: funkcja kończy swoje działanie na tej instrukcji, a dowolna wartość podana obok słowa kluczowego `return` zostaje zwrócona do kodu, z którego wywołano tę funkcję. Jak widać, bardzo przypomina to działanie instrukcji `return` znanej z większości innych języków programowania.

Zacznijmy od prostego przykładu, w którym nasza funkcja `search4vowels` będzie zwracać pojedynczą wartość. A konkretnie niech będzie to wartość `True` lub `False` w zależności od tego, czy wartość przekazana za pomocą argumentu `word` zawiera jakieś samogłoski, czy też nie.

Stanowi to co prawda pewne odejście od możliwości, jakie nasza funkcja zapewniała do tej pory, ale nie przejmuj się tym i podążaj dalej za nami, ponieważ zamierzamy wkrótce na tej podstawie zbudować coś bardziej skomplikowanego (i użytecznego). Dzięki rozpoczęciu od prostego przykładu mamy pewność, że najpierw dobrze opanujesz podstawy. Dopiero później przejdziesz do bardziej złożonych zagadnień.



Prawda jest taka...

Python oferuje wbudowaną funkcję o nazwie `bool`, która gdy otrzyma jakąkolwiek wartość, sprowadza ją do wartości `True` lub `False` i zwraca wynik tego działania.

Funkcja `bool` nie tylko operuje na dowolnych wartościach, lecz radzi sobie z dowolnymi obiektami języka Python. Wskutek tego notacja Pythona rozciąga się daleko poza prostą wartość `1` dla `True` oraz `0` dla `False`, jak ma to miejsce w innych językach programowania.

Zatrzymajmy się tu na chwilę i przyjrzyjmy się nieco dokładniej wartościom `True` oraz `False` w Pythonie, zanim powrócimy do omawiania instrukcji `return`.

Prawda z bliska

Każdy obiekt w języku Python ma odpowiednią wartość logiczną, która określa, czy obiekt ten sprowadza się do wartości `True`, czy też `False`.

Obiekt sprowadza się do wartości `False`, gdy ma wartość `0` lub `None`, jest pustym łańcuchem znakowym lub pustą wbudowaną strukturą danych. Oznacza to, że wszystkie przedstawione poniżej przykłady obiektów sprowadzają się do wartości `False`.

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False
>>> bool(None)
False
```

Jeśli obiekt sprowadza się do 0, zawsze ma logiczną wartość False.

Pusty łańcuch znakowy, pusta lista i pusty słownik sprowadzają się do wartości False.

Występująca w języku Python wartość None zawsze sprowadza się do False.

Każdy inny obiekt w Pythonie sprowadza się do wartości `True`. Poniżej znajdziesz kilka przykładów obiektów, które mają logiczną wartość `True`.

```
>>> bool(1)
True
>>> bool(-1)
True
>>> bool(42)
True
>>> bool(0.00000000000000000000000000000001)
True
>>> bool('Panic')
True
>>> bool([42, 43, 44])
True
>>> bool({'a': 42, 'b':42})
True
```

Liczba niezerowa ma zawsze logiczną wartość True, nawet gdy jest ujemna.

Liczba ta może być naprawdę mała, ale nadal jest niezerowa, dlatego sprowadza się do True.

Niepusty łańcuch znakowy ma zawsze logiczną wartość True.

Niepusta wbudowana struktura danych sprowadza się do wartości True.

Funkcji `bool` można przekazać dowolny obiekt, aby określić, czy ma on wartość logiczną `True`, czy też `False`.

Bardzo istotny jest fakt, że niepusta struktura danych sprowadza się do wartości `True`.

Zwracanie jednej wartości

Przyjrzyj się raz jeszcze przedstawionemu poniżej kodowi naszej funkcji, która w roli argumentu przyjmuje w tej chwili dowolną wartość, sprawdza ją w poszukiwaniu samogłosek, po czym wyświetla znalezione litery na ekranie.

```
def search4vowels(word):
    """Wyświetla samogłoski znalezione w słowie podanym jako argument."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Zmodyfikujemy te dwa wiersze kodu.

Wprowadzenie zmiany polegającej na tym, aby funkcja `search4vowels` zwracała wartość `True` lub `False` w zależności od tego, czy jakieś samogłoski zostaną znalezione, czy też nie, jest dość prostym zadaniem. Wystarczy, że zastąpimy dwa ostatnie wiersze kodu naszej funkcji (tworzące pętlę `for`) pojedynczym wierszem, który został przedstawiony poniżej.

```
return bool(found)
```

Wywołaj funkcję bool i...

...przekaż jej nazwę struktury danych, która zawiera wyniki wyszukiwania samogłosek.

Jeśli nie uda się nic znaleźć, nasza funkcja zwróci wartość `False`; w przeciwnym razie zwrócona zostanie wartość `True`. Po dokonaniu tej zmiany możesz przetestować nową wersję funkcji, korzystając z powłoki Pythona, aby przekonać się, co się stanie tym razem.

```
>>> search4vowels('mierni')
True
>>> search4vowels('galaktyka')
True
>>> search4vowels('hm')
False
```

Instrukcja return (dzięki zastosowaniu funkcji bool) zwraca wartość True lub False.

Podobnie jak we wcześniejszych rozdziałach, korzystamy tutaj tylko z wybranych samogłosek (a „y" do nich nie należy).

Jeśli nadal widzisz efekty działania poprzedniej wersji funkcji, sprawdź, czy zapisałeś nową wersję kodu w pliku oraz naciśnij klawisz *F5* w oknie edycji.

Porady maniaka

Nie ulegaj pokusie umieszczania w nawiasie obiektu zwracanego za pomocą instrukcji `return` do kodu wywołującego. Wcale nie musisz tego robić. Instrukcja `return` nie jest wywołaniem funkcji, a zatem użycie w tym miejscu znaków nawiasu nie jest wymogiem syntaktycznym. Możesz je oczywiście dodać (jeśli *naprawdę* Ci na tym zależy), ale większość programistów Pythona tego nie robi.

Zwracanie więcej niż jednej wartości

Funkcja z założenia zwraca pojedynczą wartość, zdarza się jednak, że zachodzi konieczność zwrócenia większej ich liczby. Jedyny sposób, który to umożliwia, polega na spakowaniu wielu wartości do pojedynczej struktury danych i zwróceniu właśnie jej. Dzięki temu formalnie nadal zwracana jest jedna rzecz, jednak potencjalnie może ona zawierać wiele oddzielnych fragmentów danych.

Tak wygląda aktualny kod naszej funkcji, która zwraca wartość logiczną (czyli pojedynczą rzecz).

Zwróć uwagę, że zaktualizowaliśmy komentarz.

```
def search4vowels(word):
    """Zwraca wartość logiczną zależną od tego, czy znalezione zostały
jakieś samogłoski."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return bool(found)
```

Zmiana, którą mamy zamiar wprowadzić, jest niezwykle prosta. Chodzi nam o to, aby zamiast wartości logicznej zwracanych było wiele wartości (zapisanych w jednym zbiorze). Wystarczy zatem tylko pozbyć się wywołania funkcji `bool`.

```
def search4vowels(word):
    """Zwraca samogłoski znalezione w słowie podanym jako argument."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return found
```

Zwróć wyniki w postaci struktury danych (zbioru).

Ponownie zaktualizowaliśmy komentarz.

W wersji programu widocznej powyżej możemy jeszcze bardziej zredukować niezbędną ilość kodu, zastępując dwa ostatnie wiersze naszej funkcji jednym i pozbywając się niepotrzebnej zmiennej `found`. Zamiast przypisywać do niej wynik działania funkcji `intersection` i zwracać ją, wystarczy, że od razu zwrócimy ten wynik.

```
def search4vowels(word):
    """Zwraca samogłoski znalezione w słowie podanym jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

Zwróć dane, nie korzystając z niepotrzebnej zmiennej found.

Nasza funkcja zwraca teraz zbiór samogłosek znalezionych w słowie, a więc dokładnie to, o co nam chodziło.

Jednak podczas testowania funkcji okazało się, że jeden ze zwracanych przez nią wyników wprawił nas w zakłopotanie...

Jazda próbna

Weźmy w obroty ostatnią wersję funkcji `search4vowels` i sprawdźmy, jakie są wyniki jej działania. Przejdź do okna edycji środowiska IDLE, w którym widoczna jest najnowsza wersja kodu, naciśnij klawisz *F5*, aby zaimportować funkcję do powłoki Pythona, po czym wywołaj ją kilka razy, aby zobaczyć, co się stanie.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
=========== RESTART: C:/Users/lsuma/Desktop/Python/R04/vsearch.py ===========
>>> search4vowels('mierni')
{'e', 'i'}
>>> search4vowels('galaktyka')
{'a'}
>>> search4vowels('życie, wszechświat i cała reszta')
{'e', 'i', 'a'}
>>> search4vowels('hm')
set()
>>>
```

Ln: 39 Col: 4

Każde z tych wywołań funkcji działa zgodnie z oczekiwaniami, choć wynik ostatniego wygląda trochę dziwnie.

O co chodzi z tym „set()"?

Każde z przykładowych wywołań funkcji działa prawidłowo w tym sensie, że przyjmuje pojedynczy argument będący łańcuchem znakowym, a następnie zwraca zbiór znalezionych samogłosek. Jeden wynik działania funkcji, będący zbiorem, może tu zawierać wiele wartości. Jednak ostatni wynik wygląda dość dziwacznie, nie sądzisz? Przyjrzyjmy mu się nieco bliżej.

Nie potrzebujemy funkcji, aby przekonać się, że słowo „hm" nie zawiera żadnych samogłosek...

```
>>> search4vowels('hm')
set()
```

...ale spójrz tylko, co zwraca nasza funkcja. Co jest grane?

Prawdopodobnie spodziewałeś się, że w wyniku działania funkcji na ekranie pojawią się znaki {} reprezentujące pusty zbiór, mamy tu jednak do czynienia z typowym nieporozumieniem, ponieważ {} oznacza pusty słownik, a *nie* pusty zbiór.

Pusty zbiór przedstawiany jest przez interpreter jako `set()`.

Może się to wydawać trochę dziwne, ale tak właśnie mają się sprawy w Pythonie. Poświęćmy chwilę na przypomnienie kilku informacji na temat wbudowanych struktur danych, zwracając szczególną uwagę na to, jak każda z nich jest reprezentowana przez interpreter, gdy jest pusta.

Przypomnienie wbudowanych struktur danych

Przypomnijmy sobie informacje na temat czterech wbudowanych struktur danych, z których możemy korzystać w języku Python. Zajmiemy się każdą z nich po kolei, omawiając listę, słownik, zbiór i na sam koniec krotkę.

Korzystając z powłoki, utwórzmy puste struktury danych za pomocą odpowiednich funkcji wbudowanych (ang. *built-in functions*, w skrótcie *BIF*), a następnie przypiszmy każdej z nich niewielką ilość danych. Po zrobieniu tego wyświetlimy zawartość każdej struktury, tak jak zostało to pokazane poniżej.

BIF to angielski skrót oznaczający funkcję wbudowaną.

```
>>> l = list()
>>> l
[]
>>> l = [ 1, 2, 3 ]
>>> l
[1, 2, 3]
```

Pusta lista

Skorzystaj z BIF o nazwie „list", aby zdefiniować pustą listę, a następnie przypisz jej jakieś dane.

```
>>> d = dict()
>>> d
{}
>>> d = { 'pierwszy': 1, 'drugi': 2, 'trzeci': 3 }
>>> d
{'pierwszy': 1, 'drugi': 2, 'trzeci': 3}
```

Pusty słownik

Skorzystaj z BIF o nazwie „dict", aby zdefiniować pusty słownik, a następnie przypisz mu jakieś dane.

```
>>> s = set()
>>> s
set()
>>> s = {1, 2, 3}
>>> s
{1, 2, 3}
```

Pusty zbiór

Skorzystaj z BIF o nazwie „set", aby zdefiniować pusty zbiór, a następnie przypisz mu jakieś dane.

Słowniki, tak samo jak zbiory, zapisuje się w nawiasach klamrowych. Puste nawiasy klamrowe są już wykorzystywane do zapisu pustego słownika, dlatego pusty słownik musi być reprezentowany w jakiś inny sposób. Oznacza się go zatem jako set().

```
>>> t = tuple()
>>> t
()
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
```

Pusta krotka

Skorzystaj z BIF o nazwie „tuple", aby zdefiniować pustą krotkę, a następnie przypisz jej jakieś dane.

Zanim przejdziemy dalej, poświęć chwilę, aby zapoznać się z przedstawionym na tej stronie sposobem, w jaki interpreter reprezentuje każdą pustą strukturę danych.

Korzystaj z adnotacji, aby poprawić swoją dokumentację

W wyniku naszego przeglądu struktur danych udało nam się potwierdzić, że funkcja `search4vowels` zwraca zbiór. Jak jednak jej użytkownicy mogą z wyprzedzeniem poznać typ wartości zwracanej oprócz tego, że mogą oczywiście wywołać funkcję i sprawdzić, jakiego typu dana jest przez nią zwracana? Skąd mają wiedzieć, czego powinni się spodziewać?

Trzeba dodać tę informację do dokumentacji funkcji. Należy w niej bardzo jasno określić, jakie są argumenty funkcji i jaka jest zwracana przez nią wartość, a także zapewnić, aby informacje te dało się łatwo znaleźć. Uzgodnienie standardu dokumentacji funkcji, który przyjąłby się wśród wszystkich programistów, okazało się dość trudne (dokument PEP 257 sugeruje jedynie *format* tekstów dokumentujących), dlatego język Python 3 obsługuje **adnotacje** (znane również jako podpowiedzi typu). Umożliwiają one dokumentowanie w standardowy sposób typu wartości zwracanego przez funkcję, jak również typów przyjmowanych przez nią argumentów. Zapamiętaj poniższe punkty:

1. **Adnotacje funkcji są opcjonalne.**
   Można ich nie używać. W gruncie rzeczy mnóstwo istniejącego kodu Pythona nie zostało w nie zaopatrzone (ponieważ możliwość dodawania adnotacji została udostępniona programistom dopiero w najnowszych wersjach Pythona 3).

2. **Adnotacje funkcji mają charakter informacyjny.**
   Zapewniają one szczegółowe informacje na temat funkcji, nie implikują jednak jakichś dodatkowych działań (takich jak sprawdzanie typów).

Skorzystajmy teraz z adnotacji, aby opisać argumenty funkcji `search4vowels`. Pierwsza z naszych adnotacji informuje, że funkcja spodziewa się argumentu `word` będącego łańcuchem znakowym (`str`), a druga wskazuje, że funkcja zwraca do kodu wywołującego wartość, której typem jest zbiór (`-> set`).

Stwierdzamy, że argument word powinien być łańcuchem znakowym.

Stwierdzamy, że funkcja zwraca zbiór do miejsca wywołania.

```
def search4vowels(word:str) -> set:
    """Wyświetla samogłoski znalezione w słowie podanym jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

Składnia adnotacji jest bardzo prosta. Po każdym argumencie dodaje się dwukropek oraz nazwę typu, który powinien mieć przekazywany argument. W naszym przykładzie ciąg `:str` określa, że funkcja spodziewa się łańcucha znakowego. Typ wartości zwracanej podaje się po liście argumentów i jest on wskazywany przez symbol strzałki, po którym występują nazwa typu i dwukropek. W naszym przykładzie ciąg `-> set` informuje, że funkcja zwraca zbiór.

Jak dotąd świetnie.

Udało nam się w standardowy sposób dodać adnotację do funkcji. Dzięki temu programiści używający naszej funkcji będą wiedzieli, czego się od nich wymaga i czego się po niej spodziewać. Jednak interpreter **nie będzie** sprawdzał, czy funkcja jest zawsze wywoływana z argumentem będącym łańcuchem znakowym, nie będzie też sprawdzał, czy zawsze zwraca ona zbiór. Co wprost prowadzi do kolejnego, dość oczywistego pytania...

Więcej szczegółów na temat adnotacji znajdziesz w dokumencie PEP 3107 znajdującym się pod adresem *https://www.python.org/dev/peps/pep-3107/*.

Dlaczego warto używać adnotacji funkcji?

Po co w ogóle zawracać sobie głowę adnotacjami, skoro interpreter Pythona nie zamierza ich wykorzystywać do sprawdzania typów argumentów funkcji i zwracanej przez nią wartości?

Celem adnotacji *nie* jest ułatwianie pracy interpreterowi, lecz użytkownikowi funkcji. Adnotacje stanowią **standard dokumentacji**, a nie rodzaj mechanizmu wymuszania typów.

W gruncie rzeczy interpreter zupełnie nie dba o typy Twoich argumentów, nie obchodzi go też typ wartości zwracanej przez funkcję. Wywołuje on Twoją funkcję z takimi argumentami, jakie zostaną podane (bez względu na typy, które mają), wykonuje jej kod, a następnie zwraca do miejsca wywołania wartość przekazaną instrukcji `return` bez względu na to, czym jest ta wartość. Typy danych przekazywanych w obydwu kierunkach nie są w żaden sposób analizowane przez interpreter.

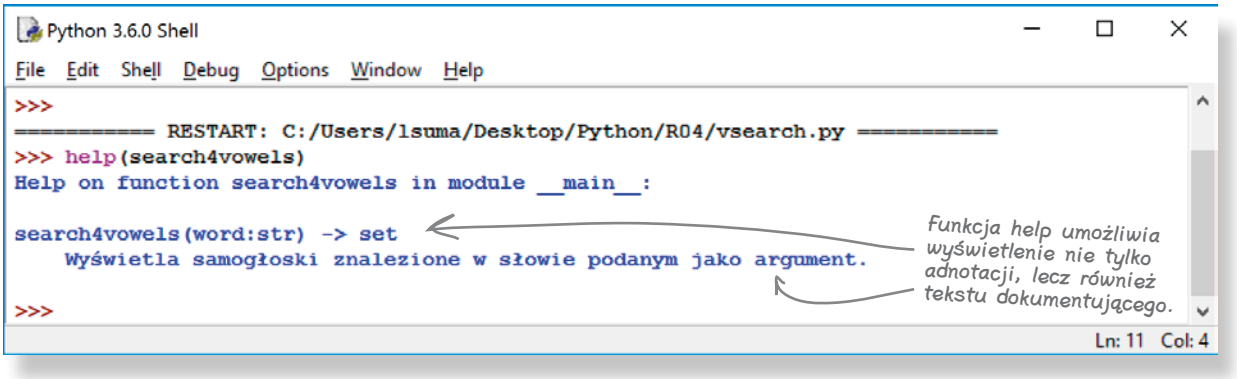
Adnotacje umożliwiają programistom korzystającym z Twojej funkcji uniknięcie konieczności czytania jej kodu w celu poznania typów danych przyjmowanych i zwracanych przez tę funkcję. A właśnie to muszą oni robić, gdy nie używasz adnotacji. Nawet najlepiej napisany tekst dokumentujący nadal trzeba będzie czytać, jeśli do funkcji nie doda się adnotacji.

To z kolei prowadzi do następnego pytania: jak możemy się zapoznać z adnotacją, nie czytając kodu funkcji? Sprawdź to, naciskając klawisz *F5* w oknie edycji środowiska IDLE, a następnie korzystając z wbudowanej funkcji `help` po znaku zachęty >>>.

Korzystaj z adnotacji, aby dokumentować swoje funkcje. Używaj wbudowanej funkcji help, aby przeglądać adnotacje.

Jazda próbna

Jeśli nie zrobiłeś tego do tej pory, skorzystaj teraz z edytora środowiska IDLE i dodaj adnotację do Twojej kopii funkcji `search4vowels`, zapisz kod w pliku i naciśnij klawisz *F5*. Powłoka Pythona zostanie zrestartowana, a znak zachęty >>> będzie oczekiwał na Twoje działanie. Poproś wbudowaną funkcję `help` o wyświetlenie dokumentacji funkcji `search4vowels`, tak jak pokazano to poniżej.



Funkcje: co już wiemy

Zatrzymajmy się na chwilę i dokonajmy przeglądu tego, czego dowiedzieliśmy się (do tej pory) na temat funkcji w języku Python.

CELNE SPOSTRZEŻENIA

- Funkcje to nazwane fragmenty kodu.
- Słowo kluczowe `def` wykorzystuje się, aby nadać funkcji nazwę. Kod tworzący funkcję powinien znajdować się poniżej i być odpowiednio wcięty w odniesieniu do słowa `def`.
- Łańcuchy znakowe umieszczone pomiędzy potrójnymi znakami cudzysłowu mogą być wykorzystywane do dodawania wielowierszowych komentarzy do funkcji. Jeśli są one używane w ten sposób, określa się je jako *teksty dokumentujące*.
- Funkcje mogą przyjmować dowolną liczbę nazwanych argumentów; jednak argumentów tych może również wcale nie być.
- Instrukcja `return` umożliwia funkcji zwracanie dowolnej liczby wartości; funkcja może również nie zwracać żadnych danych.
- Adnotacje funkcji mogą być używane do dokumentowania typów przyjmowanych argumentów, jak również typu wartości zwracanej przez funkcję.

Poświęćmy chwilę na przejrzenie przedstawionego poniżej kodu funkcji `search4vowels` po raz kolejny. Teraz, gdy przyjmuje ona argument i zwraca zbiór, jest o wiele bardziej użyteczna, niż była w swojej pierwszej wersji, którą zaprezentowaliśmy na początku tego rozdziału, ponieważ w tej chwili można ją wykorzystać w znacznie większej liczbie miejsc.

Najnowsza wersja naszej funkcji

```
def search4vowels(word:str) -> set:
    """Wyświetla samogłoski znalezione w słowie podanym jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

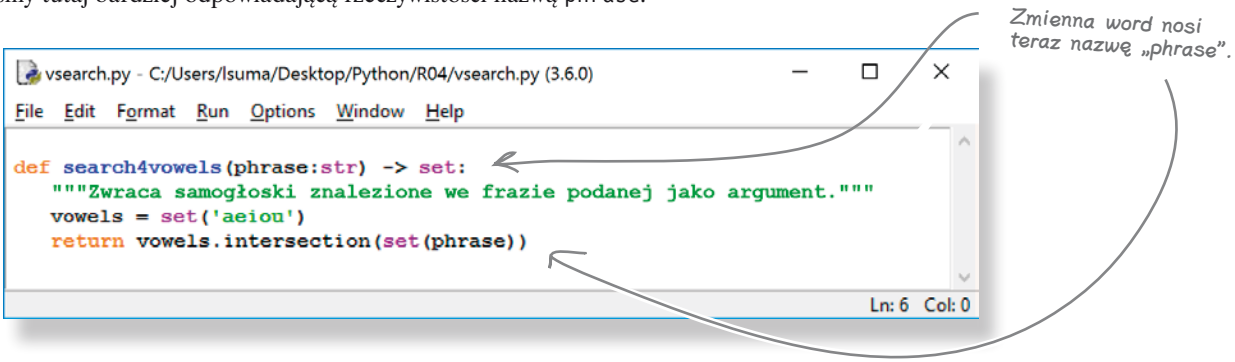
Funkcja ta byłaby jeszcze bardziej użyteczna, gdyby poza tym, że przyjmuje w roli argumentu słowa, które należy przeszukać, przyjmowała również drugi argument, określający, czego należy w danym słowie szukać. Umożliwiłoby nam to wyszukiwanie liter należących do dowolnego zbioru, nie tylko tych pięciu samogłosek, z którymi mieliśmy do czynienia do tej pory.

Poza tym korzystanie ze słowa `word` jako nazwy argumentu jest w porządku, ale nie jest to rozwiązanie idealne, ponieważ funkcja w oczywisty sposób przyjmuje jako swój argument *dowolny* łańcuch znakowy, a nie jedynie pojedyncze słowo, jak sugeruje to ta nazwa. Lepszą nazwą zmiennej byłoby `phrase`, ponieważ słowo „fraza” bardziej precyzyjnie opisuje, co spodziewamy się otrzymać od użytkowników naszej funkcji.

Zmieńmy naszą funkcję w taki sposób, aby uwzględnić tę ostatnią sugestię.

Tworzenie funkcji ogólnego przeznaczenia

Poniżej znajdziesz wersję funkcji `search4vowels` (widoczną w oknie edycji środowiska IDLE), która została zmodyfikowana w taki sposób, aby uwzględniać drugą z sugestii zamieszczonych na końcu poprzedniej strony. A konkretnie: nazwę zmiennej `word` zastąpiliśmy tutaj bardziej odpowiadającą rzeczywistości nazwą `phrase`.



Pierwszą sugestią zamieszczoną na końcu poprzedniej strony było to, aby umożliwić użytkownikom określanie zbioru liter, które mają być wyszukiwane, zamiast zawsze używać zapisanych na sztywno pięciu samogłosek. Aby to zrobić, możemy dodać drugi argument funkcji, określający, których liter należy szukać w łańcuchu znakowym `phrase`. Zmiana ta jest dość prosta do zrobienia. Jednak gdy ją wprowadzimy, funkcja będzie miała niewłaściwą nazwę, ponieważ nie będzie już wyszukiwała samogłosek, lecz dowolne litery należące do podanego zbioru. Zamiast zmieniać bieżącą funkcję, na jej bazie utwórzmy nową. Oto co proponujemy Ci zrobić:

1. **Nadaj funkcji bardziej ogólną nazwę.**
   Zamiast kontynuować poprawianie funkcji `search4vowels`, utwórzmy nową funkcję o nazwie `search4letters`, która znacznie lepiej oddaje przeznaczenie tej funkcji.

2. **Dodaj drugi argument.**
   Dodanie drugiego argumentu umożliwi nam określanie zbioru liter, które mają być wyszukiwane w łańcuchu znakowym. Temu drugiemu argumentowi nadajmy nazwę `letters`. Nie zapomnijmy przy tym również o dodaniu do niego odpowiedniej adnotacji.

3. **Usuń zmienną vowels.**
   Użycie nazwy `vowels` w zestawie funkcji nie ma już najmniejszego sensu, ponieważ od tej pory będziemy wyszukiwać litery podane przez użytkownika.

4. **Zaktualizuj tekst dokumentujący.**
   Nie ma sensu kopiować, a następnie zmieniać kodu, jeśli odpowiednio nie zmodyfikujemy też tekstu dokumentującego. Nasza dokumentacja musi zostać zmieniona w taki sposób, aby odzwierciedlać działanie nowej funkcji.

Wykonamy te cztery zadania wspólnie. Pamiętaj, aby zmieniać zawartość pliku *vsearch.py* wraz z tym, jak będziemy omawiać poszczególne punkty, aby Twój kod odpowiednio odzwierciedlał prezentowane przez nas modyfikacje.

Tworzenie kolejnej funkcji (1. z 3)

Jeśli nie zrobiłeś tego wcześniej, otwórz teraz plik *vsearch.py* w oknie edycji środowiska IDLE.

Krok 1. polega na utworzeniu nowej funkcji, którą nazwiemy `search4letters`. Miej świadomość, że wskazówki zawarte w dokumencie PEP 8 sugerują, aby wszystkie funkcje najwyższego poziomu były otoczone dwoma pustymi wierszami. Wszystkie pliki z kodem, które możesz pobrać z internetowej strony związanej z tą książką (*ftp://ftp.helion.pl/przyklady/pytrg2.zip*), spełniają to wymaganie, ale nie jest tak w przypadku kodu przedstawionego na jej drukowanych stronach (ponieważ bardzo zależy nam tu na oszczędności miejsca).

Na końcu pliku napisz słowo kluczowe ***def***, a następnie nazwę swojej nowej funkcji, tak jak zostało to przedstawione poniżej.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels(phrase:str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters
```

Zacznij od nadania nazwy swojej nowej funkcji.

Ln: 7 Col: 18

W **kroku 2.** uzupełnimy wiersz `def` naszej funkcji, dodając do niego nazwy dwóch wymaganych argumentów, czyli `phrase` i `letters`. Pamiętaj, aby listę argumentów umieścić w nawiasie, nie zapomnij też o znaku dwukropka występującym po każdej nazwie argumentu.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels(phrase:str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    |
```

Podaj listę argumentów i nie zapomnij o znaku dwukropka (oraz o odpowiedniej adnotacji).

Ln: 8 Col: 3

Czy zauważyłeś, że edytor środowiska IDLE przewidział, że następny wiersz kodu powinien być wcięty (i automatycznie ustawił kursor w odpowiednim miejscu)?

Po wykonaniu dwóch pierwszych kroków jesteśmy już gotowi do napisania kodu funkcji. Będzie on dość podobny do tego, który tworzy funkcję `search4vowels`, z tym że planujemy się tu pozbyć zależności od zmiennej `vowels`.

Tworzenie kolejnej funkcji (2. z 3)

W **kroku 3.** napiszemy kod naszej nowej funkcji w taki sposób, aby uniknąć konieczności korzystania w niej ze zmiennej `vowels`. Moglibyśmy wciąż używać tej zmiennej, ale nadać jej inną nazwę (jako że nazwa `vowels` nie oddaje już właściwie znaczenia przechowywanych w niej danych), jednak zmienna tymczasowa nie jest nam tak naprawdę już tu potrzebna, mniej więcej z tego samego powodu, dla którego wcześniej przestaliśmy używać zmiennej `found`. Spójrz na widoczny poniżej nowy wiersz kodu należący do funkcji `search4letters`, który spełnia to samo zadanie, jakie wypełniały wcześniej odpowiednie dwa wiersze funkcji `search4vowels`.

Dwa wiersze kodu zostały zastąpione jednym.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels(phrase:str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    return set(letters).intersection(set(phrase))
```

Ln: 9 Col: 0

Jeśli ten pojedynczy wiersz kodu funkcji `search4letters` sprawia, że drapiesz się z zakłopotaniem w głowę, nie popadaj w rozpacz. Wygląda on na znacznie bardziej skomplikowany, niż jest w rzeczywistości. Przyjrzyjmy się temu wierszowi w skupieniu i wyjaśnijmy, co się tu dokładnie dzieje. Na początku wartość argumentu `letters` zostaje zmieniona w zbiór.

```
set(letters)
```

Utwórz obiekt zbioru ze zmiennej letters.

To wywołanie wbudowanej funkcji `set` skutkuje utworzeniem obiektu zbioru z liter wchodzących w skład łańcucha znakowego przechowywanego przez zmienną `letters`. Obiektu tego nie musimy przypisywać do żadnej zmiennej, ponieważ jesteśmy bardziej zainteresowani natychmiastowym skorzystaniem ze zbioru liter niż przechowywaniem go w zmiennej w celu późniejszego użycia. Aby skorzystać z właśnie utworzonego obiektu zbioru, należy dodać po nim kropkę, a następnie wskazać metodę, którą chcemy wywołać, ponieważ nawet obiekty, które nie zostały przypisane do zmiennych, zapewniają odpowiednie metody. Jak już wiemy dzięki zastosowaniu zbiorów w poprzednim rozdziale, metoda `intersection` bierze zbiór znaków zapisany w swoim argumencie (`phrase`) i wyznacza jego iloczyn zbiorów z istniejącym zbiorem obiektów, na rzecz którego została wywołana (`letters`).

```
set(letters).intersection(set(phrase))
```

Przeprowadź operację iloczynu zbiorów na zbiorze obiektów utworzonym ze zmiennej letters i zbiorze obiektów utworzonym ze zmiennej phrase.

Na koniec uzyskany wynik iloczynu zbioru zostaje zwrócony do wywołującego kodu jako wynik działania funkcji. Umożliwia to instrukcja `return`.

```
return set(letters).intersection(set(phrase))
```

Odeślij wyniki z powrotem do kodu wywołującego funkcję.

Tworzenie kolejnej funkcji (3. z 3)

Został nam już do wykonania tylko **krok 4.**, który polega na dodaniu do naszej nowo utworzonej funkcji tekstu dokumentującego. Aby to zrobić, wprowadź łańcuch znakowy umieszczony w potrójnych znakach cudzysłowu tuż po wierszu zawierającym słowo kluczowe `def`. Poniżej możesz zobaczyć, co napisaliśmy my (nasz komentarz jest krótki, ale świetnie spełnia swoją rolę).

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels(phrase:str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    """Zwraca zbiór liter ze zmiennej letters znalezionych w zmiennej phrase."""
    return set(letters).intersection(set(phrase))
```

Ln: 10 Col: 0

Tekst dokumentujący

Gdy to już zrobiliśmy, nasze cztery kroki zostały wykonane, a funkcja `search4letters` może zostać poddana testom.

Po co zadawać sobie tyle trudu, aby utworzyć jednowierszową funkcję? Czy nie byłoby lepiej po prostu skopiować i wkleić ten jeden wiersz kodu wszędzie tam, gdzie jest potrzebny?

Funkcje mogą również ukrywać złożoność.

To prawda, że właśnie utworzyliśmy funkcję, która zawiera zaledwie jeden wiersz kodu, co może się wydawać trochę bezcelowe, jeśli za wyznacznik przyjmiemy tu wąsko rozumianą „oszczędność” pracy czy miejsca. Zwróć jednak uwagę, że nasza funkcja zawiera dość skomplikowany wiersz kodu, który został ukryty przed używającymi jej osobami i właśnie to może okazać się powodem, dla którego warto zadawać sobie cały ten trud (nie wspominając już, że jest to rozwiązanie znacznie lepsze niż wielokrotne kopiowanie i wklejanie fragmentu kodu).

Większość programistów będzie w stanie domyślić się, co robi funkcja `search4letters`, gdy tylko natrafi na jej wywołanie w jakimś programie. Gdyby jednak Ci sami ludzie znaleźli ten pojedynczy skomplikowany wiersz kodu, prawdopodobnie drapaliby się długo po głowie i usilnie zastanawiali, na czym polega jego działanie. A zatem mimo że funkcja `search4letters` jest „krótka”, utworzenie jej jest nadal dobrym pomysłem, ponieważ umożliwia wydobycie i ukrycie całej tej niezbędnej złożoności za pomocą funkcji.

Jazda próbna

Zapisz plik *vsearch.py* raz jeszcze, a następnie naciśnij klawisz *F5*, aby wypróbować działanie funkcji `search4letters`.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
========== RESTART: C:/Users/lsuma/Desktop/Python/R04/vsearch.py ==========
>>> help(search4letters)
Help on function search4letters in module __main__:

search4letters(phrase:str, letters:str) -> set
    Zwraca zbiór liter ze zmiennej letters znalezionych w zmiennej phrase.

>>> search4letters('mierni', 'aeiou')
{'e', 'i'}
>>> search4letters('galaktyka', 'tym')
{'t', 'y'}
>>> search4letters('życie, wszechświat i cała reszta', 'o')
set()
>>>
                                                          Ln: 31  Col: 4
```

Użyj wbudowanej funkcji help, aby dowiedzieć się, jak korzystać z funkcji search4letters.

Wszystkie te przykładowe wywołania funkcji zwróciły wyniki, których się spodziewaliśmy.

Funkcja `search4letters` ma teraz bardziej ogólny charakter niż funkcja `search4vowels`, ponieważ przyjmuje *dowolny* zbiór liter i wyszukuje je w podanym łańcuchu znakowym, zamiast wyszukiwać w nim wyłącznie samogłoski `a`, `e`, `i`, `o` oraz `u`. Powoduje to, że nasza nowa funkcja jest znacznie bardziej użyteczna niż funkcja `search4vowels`. Wyobraźmy sobie teraz, że mamy ogromną, istniejącą już bazę kodu, w której bardzo intensywnie wykorzystywana była oryginalna funkcja `search4vowels`. Podjęto decyzję o zaprzestaniu używania tej starej funkcji i zastąpieniu jej funkcją `search4letters`, ponieważ „ktoś na górze" uznał, że nie ma potrzeby utrzymywania obydwu tych funkcji, skoro `search4letters` może swobodnie robić to, za co do tej pory odpowiadała funkcja `search4vowels`. Globalna operacja wyszukania nazwy „search4vowels" i zastąpienia wszystkich jej wystąpień za pomocą nazwy „search4letters" w całej bazie kodu nie załatwi nam tu sprawy, ponieważ do każdego wywołania funkcji trzeba jeszcze dodać wartość drugiego argumentu, którym dla funkcji `search4letters` symulującej działanie funkcji `search4vowels` jest ciąg `aeiou`. A zatem przedstawione poniżej przykładowe wywołanie jednoargumentowe:

```
search4vowels("Podaj jajko!")
```

powinno zostać zastąpione dwuargumentowym wywołaniem o (trudnej do automatycznej zamiany) postaci:

```
search4letters("Podaj jajko!", 'aeiou')
```

Byłoby miło, gdyby w jakiś sposób dało się tu ustawić *domyślną wartość* dla drugiego argumentu funkcji `search4letters`, dzięki czemu mogłaby ona używać tej wartości, gdyby żadna inna nie została podana. Gdybyśmy mogli tak ustawić tę domyślą wartość, aby był to ciąg `aeiou`, moglibyśmy skorzystać z globalnej operacji wyszukania starej funkcji i zastąpienia jej wszystkich wystąpień nową (a to jest już bardzo proste do zrobienia).

Czyż nie byłoby cudownie, gdyby Python pozwalał mi określać domyślne wartości argumentów? Wiem jednak, że to tylko fantazja...

Określanie domyślnych wartości argumentów

Każdemu argumentowi funkcji można w języku Python przypisać domyślną wartość, która zostanie automatycznie użyta, gdy kod wywołujący tę funkcję nie dostarczy własnej. Mechanizm przypisywania argumentowi wartości domyślnej działa w bardzo prosty sposób. Aby z niego skorzystać, wystarczy umieścić odpowiednie przypisanie w wierszu `def` naszej funkcji.

Poniżej został przedstawiony wiersz `def` bieżącej wersji funkcji `search4letters`.

```
def search4letters(phrase:str, letters:str) -> set:
```

Ta wersja funkcji spodziewa się otrzymać *dokładnie* dwa argumenty, z których jeden będzie stanowił wartość zmiennej `phrase`, a drugi — wartość zmiennej `letters`. Jeśli temu ostatniemu przypiszemy wartość domyślną, wiersz `def` naszej funkcji przybierze postać pokazaną poniżej.

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Argumentowi letters została tu przypisana wartość domyślna, która będzie używana zawsze wtedy, gdy wywołujący tę funkcję kod nie dostarczy własnej wartości.

Z funkcji `search4letters` nadal możemy korzystać tak, jak miało to miejsce wcześniej, tj. podając wartości obydwu argumentów, tam gdzie jest to konieczne. Gdy jednak zapomnimy o podaniu drugiego argumentu (`letters`), interpreter przypisze mu w naszym imieniu wartość `aeiou`.

Gdybyśmy wprowadzili tę zmianę w kodzie naszej funkcji umieszczonym w pliku *vsearch.py* (i zapisali go na dysku), moglibyśmy wywołać nasze funkcje tak, jak zostało to pokazane poniżej.

```
>>> search4letters('życie, wszechświat i cała reszta')
{'i', 'a', 'e'}
>>> search4letters('życie, wszechświat i cała reszta', 'aeiou')
{'i', 'a', 'e'}
>>> search4vowels('życie, wszechświat i cała reszta')
{'i', 'a', 'e'}
```

Te trzy wywołania funkcji zwracają takie same wyniki.

Tutaj wywołujemy funkcję search4vowels, nie funkcję search4letters.

Wywołania te nie tylko zwracają te same wyniki, lecz wyraźnie pokazują również, że funkcja `search4vowels` nie jest nam już do niczego potrzebna, ponieważ argument `letters` funkcji `search4letters` ma swoją wartość domyślną (jeśli chcesz się o tym przekonać, porównaj pierwsze widoczne powyżej wywołanie z ostatnim).

Gdybyśmy zostali poproszeni o pozbycie się funkcji `search4vowels` z programu i zastąpienie wszystkich jej wywołań w naszej bazie kodu wywołaniami funkcji `search4letters`, dzięki wykorzystaniu mechanizmu wartości domyślnej argumentu moglibyśmy to zrobić za pomocą prostej operacji globalnego wyszukania i zastąpienia odpowiedniej nazwy. Co więcej, nie musimy używać funkcji `search4letters` wyłącznie do wyszukiwania samogłosek. Drugi argument funkcji umożliwia nam podanie *dowolnego* zbioru znaków, które należy znaleźć. Dzięki temu funkcja `search4letters` ma bardziej ogólny charakter *i* jest dużo użyteczniejsza.

Układ pozycyjny kontra słowa kluczowe

Jak się właśnie przekonaliśmy, funkcja `search4letters` może być wywoływana z jednym lub dwoma argumentami, przy czym drugi argument jest opcjonalny. Jeśli podasz tylko jeden argument, argument `letters` przyjmie domyślną wartość będącą łańcuchem znakowym zawierającym samogłoski. Przyjrzyj się raz jeszcze pokazanemu poniżej wierszowi `def` naszej funkcji.

Wiersz def naszej funkcji

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Interpreter Pythona zapewnia nie tylko obsługę wartości domyślnych argumentów, lecz również umożliwia wywołanie funkcji z **argumentami ze słowami kluczowymi**. Aby zrozumieć tę koncepcję, zastanów się, przyglądając się przedstawionemu poniżej przykładowi, jak do tej pory wywoływaliśmy funkcję `search4letters`.

```
search4letters('galaktyka', 'tym')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

W widocznym powyżej wywołaniu dwa podane łańcuchy znakowe są przypisywane, odpowiednio, do argumentów `phrase` i `letters` na podstawie swoich pozycji. Oznacza to, że pierwszy łańcuch zostaje przypisany do zmiennej `phrase`, drugi zaś — do zmiennej `letters`. Sposób ten znany jest pod nazwą **przypisania pozycyjnego**, ponieważ wykorzystuje się w jego przypadku kolejność argumentów.

W języku Python możliwe jest jednak również odwoływanie się do argumentów za pomocą ich nazw. Jeśli korzystasz z tego sposobu, kolejność występowania wartości przestaje mieć znaczenie. Technika ta nosi nazwę **przypisania za pomocą słowa kluczowego**. Aby z niej skorzystać, podczas wywoływania funkcji powinieneś *w dowolnej kolejności* przypisać każdy z łańcuchów znakowych do odpowiedniego argumentu wskazanego za pomocą nazwy, co zaprezentowano poniżej.

Kolejność argumentów nie ma znaczenia, jeśli w wywołaniu funkcji korzysta się ze słów kluczowych.

```
search4letters(letters='tym', phrase='galaktyka')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

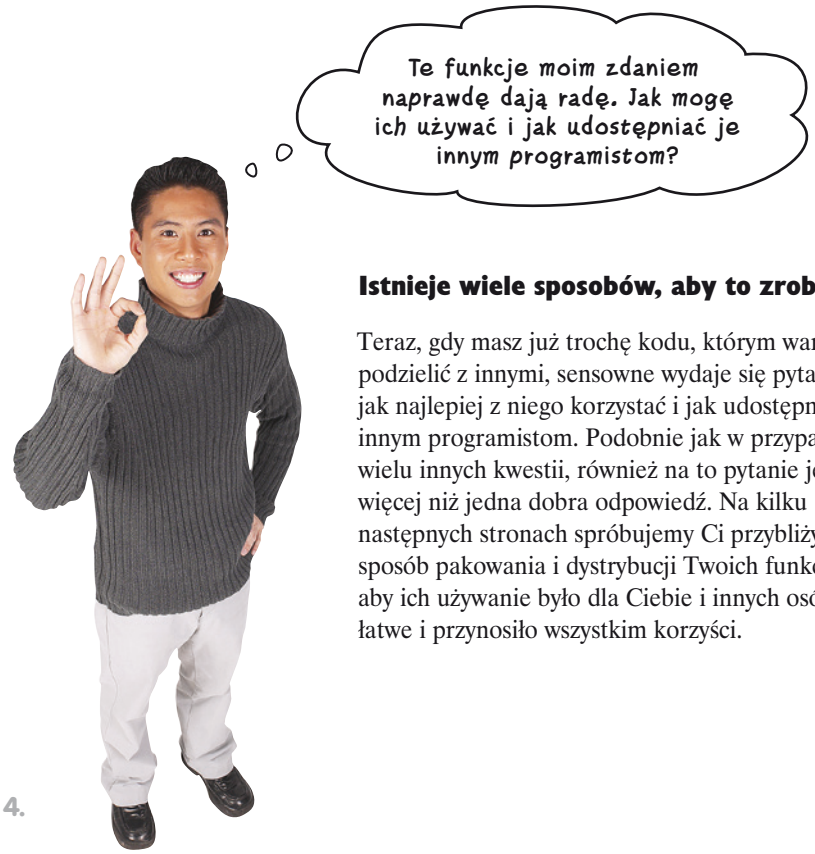
Obydwa przedstawione na tej stronie wywołania funkcji `search4letters` odnoszą ten sam skutek, zwracając zbiór zawierający litery `t` oraz `y`. Choć w przypadku naszej niewielkiej funkcji może być dość trudno dostrzec zalety korzystania z argumentów ze słowami kluczowymi, rozwiązanie to oferuje dużą elastyczność, której znaczenie staje się jasne, gdy wywołuje się funkcję przyjmującą bardzo wiele argumentów. Przykład tego rodzaju funkcji (należącej do biblioteki standardowej) poznasz jeszcze przed końcem tego rozdziału.

Odświeżanie tego, co już wiemy o funkcjach

Przypomnijmy sobie, co nowego dowiedziałeś się na temat funkcji do tej pory, gdy poświęciłeś już nieco czasu na poznawanie sposobu działania argumentów.

CELNE SPOSTRZEŻENIA

- Oprócz tego, że funkcje wspierają ponowne użycie kodu, mogą też ukrywać złożoność. Jeśli masz w swoim kodzie jakiś skomplikowany wiersz, z którego będziesz często korzystał, wydziel go i ukryj za prostym wywołaniem funkcji.
- Dowolny argument funkcji może mieć wartość domyślną, którą podaje się w wierszu `def`. Jeśli tak jest, podawanie wartości takiego argumentu przy wywoływaniu funkcji ma charakter opcjonalny.
- Argumenty możesz przypisywać w sposób pozycyjny, lecz możesz też korzystać ze słów kluczowych. Jeśli używasz tej drugiej metody, dopuszczalna jest dowolna kolejność argumentów (ponieważ wszelkie prawdopodobieństwo wystąpienia niejednoznaczności zostaje wyeliminowane dzięki zastosowaniu słów kluczowych i pozycje argumentów przestają mieć jakiekolwiek znaczenie)..



Istnieje wiele sposobów, aby to zrobić.

Teraz, gdy masz już trochę kodu, którym warto się podzielić z innymi, sensowne wydaje się pytanie, jak najlepiej z niego korzystać i jak udostępniać go innym programistom. Podobnie jak w przypadku wielu innych kwestii, również na to pytanie jest więcej niż jedna dobra odpowiedź. Na kilku następnych stronach spróbujemy Ci przybliżyć taki sposób pakowania i dystrybucji Twoich funkcji, aby ich używanie było dla Ciebie i innych osób łatwe i przynosiło wszystkim korzyści.

Funkcje tworzą moduły

moduł

Po przejściu całej tej drogi związanej z tworzeniem funkcji wielokrotnego użytku (a właściwie dwóch takich funkcji, jak ma to miejsce w przypadku kodu znajdującego się w pliku *vsearch.py*), warto zapytać: *w jaki sposób najlepiej udostępniać swoje funkcje?*

Możliwe jest oczywiście udostępnianie funkcji przez kopiowanie i wklejanie jej kodu wszędzie tam, gdzie jest on potrzebny, ale z uwagi na to, że jest to bardzo zła i nieefektywna metoda, nie będziemy jej brać pod uwagę. Wiele kopii tej samej funkcji powoduje szybkie zaśmiecenie bazy kodu i jest gwarantowaną drogą do katastrofy (zwłaszcza że z pewnością nadejdzie kiedyś taki moment, gdy będziesz musiał zmienić sposób działania swojej funkcji). O wiele lepszym pomysłem jest utworzenie **modułu** zawierającego pojedynczą, kanoniczną kopię wszystkich funkcji, które chcesz udostępniać. To z kolei prowadzi do następnego pytania: *jak tworzy się moduły w Pythonie?*

Udostępniaj swoje funkcje w modułach.

Odpowiedź na to pytanie nie może być prostsza: modułem jest każdy plik zawierający funkcje. Szczęśliwie dla nas oznacza to również, że plik *vsearch.py* stanowi *już* moduł. Poniżej możesz go znów ujrzeć, tym razem w pełni jego modułowej chwały.

vsearch.py - C:/Users/lsuma/Desktop/Python/R04/vsearch.py (3.6.0)

File Edit Format Run Options Window Help

```
def search4vowels(phrase:str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    """Zwraca zbiór liter ze zmiennej letters znalezionych w zmiennej phrase."""
    return set(letters).intersection(set(phrase))
```

Ln: 10 Col: 0

Plik vsearch.py zawiera funkcje, co czyni go pełnoprawnym modułem.

Tworzenie modułów nie może być prostsze, jednak...

Tworzenie modułów to bułka z masłem: wystarczy tylko utworzyć plik zawierający funkcje, które chcesz udostępnić.

Gdy moduł już istnieje, udostępnienie jego zawartości swoim programom jest również bardzo prostym działaniem: musisz tu jedynie zaimportować ten moduł za pomocą instrukcji `import` języka Python.

Samo to działanie nie jest skomplikowane. Powinieneś jednak wiedzieć, że interpreter zakłada, iż moduł, który importujesz, znajduje się w miejscu wskazywanym przez **ścieżkę wyszukiwania** (ang. *search path*), a zapewnienie, żeby tak właśnie było, może wymagać nieco więcej zachodu. Na kilku kolejnych stronach przedstawimy Ci tajniki importowania modułu.

Jak znajduje się moduły?

moduł

Przypomnij sobie z pierwszego rozdziału tej książki, jak importowaliśmy funkcję `randint` należącą do modułu `random`, który jest dostarczany w ramach biblioteki standardowej języka Python, a następnie jak korzystaliśmy z tej funkcji. Poniżej możesz zobaczyć, co robiliśmy wtedy w powłoce.

Wskaż moduł, który ma zostać zaimportowany, a następnie...

```
>>> import random
>>> random.randint(0, 255)
42
```

...wywołaj jedną z funkcji należących do tego modułu.

To, co się dzieje podczas importowania modułu, jest bardzo szczegółowo opisane w dokumentacji Pythona, w którą możesz się w dowolnej chwili zagłębić, jeśli brak wiedzy na ten temat mocno Ci doskwiera. Jednak wszystkim, co naprawdę jest Ci teraz potrzebne, jest informacja na temat trzech najważniejszych miejsc, które interpreter sprawdza w poszukiwaniu modułu. Są to:

1. **Twój bieżący katalog roboczy.**
   Jest to folder, w którym zdaniem interpretera obecnie pracujesz.

2. **Lokalizacje pakietów interpretera.**
   Są to katalogi zawierające wszystkie pochodzące od osób trzecich moduły Pythona, które mogłeś zainstalować (w tym wszystkie te, które sam napisałeś).

3. **Lokalizacje biblioteki standardowej.**
   Są to katalogi zawierające wszystkie moduły, które tworzą bibliotekę standardową.

Porady maniaka

W zależności od tego, z którego systemu operacyjnego korzystasz, nazwą, którą określa się miejsce przechowywania plików, może być **katalog** (ang. *directory*) lub **folder**. W książce tej będziemy się posługiwali terminem „folder" z wyjątkiem sytuacji, gdy będzie mowa o *bieżącym katalogu roboczym* (który na dobre zagościł już w naszym słowniku informatycznym).

Kolejność, w jakiej interpreter przeszukuje miejsca wymienione w punktach 2. i 3., może się różnić w zależności od wielu czynników. Nie martw się jednak, ponieważ nie musisz wiedzieć, jak dokładnie działa ten mechanizm wyszukiwania. To, co *jest* istotne, to fakt, że interpreter *najpierw* przeszukuje bieżący katalog roboczy, co może doprowadzić do pewnych kłopotów, gdy będziesz korzystał z własnych, niestandardowych modułów.

Aby zademonstrować, co może pójść nie tak, wykonajmy pewne ćwiczenie, które zostało przygotowane specjalnie po to, aby uwypuklić ten problem. Oto co powinieneś zrobić, zanim zaczniemy:

- ☐ Utwórz folder o nazwie *mymodules*, którego będziemy używać do przechowywania naszych modułów. Nie ma najmniejszego znaczenia, gdzie w swoim systemie plików umieścisz ten folder; po prostu utwórz go w miejscu, w którym masz uprawnienia do odczytywania i zapisywania danych.
- ☐ Przenieś swój plik *vsearch.py* do tego nowo utworzonego folderu. W folderze *mymodules* powinna się znajdować jedyna kopia pliku *vsearch.py* na Twoim komputerze.

Uruchamianie Pythona z wiersza poleceń

moduł

Zamierzamy teraz uruchomić interpreter Pythona z poziomu wiersza poleceń (lub terminala) Twojego systemu operacyjnego, aby zademonstrować, co może tu pójść nie tak (choć problem, który chcemy omówić, zdarza się również w środowisku IDLE).

Jeśli korzystasz z dowolnej wersji systemu *Windows*, otwórz okno wiersza poleceń i wykonuj kolejne opisane poniżej kroki. Jeśli nie używasz *Windowsa*, to wiedz, że więcej szczegółów na temat swojej platformy znajdziesz w połowie następnej strony (ale mimo wszystko kontynuuj lekturę bieżącej). Interpreter Pythona możesz wywołać, wpisując po znaku zachęty `C:\>` wiersza poleceń systemu *Windows* (czyli poza środowiskiem IDLE) polecenie ***py -3***. Zauważ poniżej, że przed uruchomieniem interpretera skorzystaliśmy z polecenia `cd`, aby przejść do folderu *mymodules* i uczynić go naszym bieżącym katalogiem roboczym. Zwróć też uwagę na to, że interpreter można w każdej chwili opuścić, używając polecenia `quit()` po znaku zachęty >>>.

Przejdź do folderu o nazwie „mymodules".

Uruchom interpreter języka Python 3.

Zaimportuj swój moduł.

Użyj funkcji należących do modułu.

Opuść interpreter Pythona i wróć do wiersza poleceń swojego systemu operacyjnego.

Wiersz polecenia

```
C:\Users\lsuma\Desktop\Python>cd mymodules

C:\Users\lsuma\Desktop\Python\mymodules>py -3
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('mierni')
{'i', 'e'}
>>> vsearch.search4letters('galaktyka', 'tym')
{'t', 'y'}
>>> quit()

C:\Users\lsuma\Desktop\Python\mymodules>
```

Wszystko działa tu zgodnie z oczekiwaniami. Udało nam się bez problemu zaimportować moduł `vsearch`, a następnie użyć każdej z jego funkcji, umieszczając w tym celu nazwę modułu i kropkę przed nazwą funkcji. Zauważ, że znak zachęty >>> zachowuje się dokładnie tak samo w systemowym wierszu poleceń, jak zachowywał się w oknie środowiska IDLE (jedyną różnicą jest tu brak wyróżniania składni za pomocą kolorów). Jakby nie było, w końcu to ten sam interpreter Pythona.

Choć przedstawiony powyżej przykład użycia interpretera pokazał, że wszystko działa tak, jak powinno, musisz wiedzieć, że jest tak wyłącznie dlatego, iż zaczęliśmy z niego korzystać z folderze zawierającym plik *vsearch.py*. Dzięki temu folder ten stał się naszym bieżącym katalogiem roboczym. Mając świadomość sposobu, w jaki interpreter poszukuje modułów, wiemy, że bieżący katalog roboczy przeszukiwany jest najpierw, dlatego nie powinno nas szczególnie zaskoczyć, że nasza współpraca z interpreterem przebiegła pomyślnie i że udało mu się bez problemu znaleźć nasz moduł.

Co się jednak stanie, gdy nasz moduł nie będzie się znajdował w bieżącym katalogu roboczym?

Nieodnalezienie modułów powoduje wystąpienie błędów importowania

moduł

Powtórz ćwiczenie przedstawione na poprzedniej stronie, gdy opuścisz folder zawierający nasz moduł. Sprawdźmy, co się stanie, gdy teraz spróbujemy zaimportować moduł. Poniżej zostało zamieszczone okno wiersza poleceń systemu *Windows* z widocznym efektem naszej próby.

Przejdź do innego folderu (tutaj przechodzimy do katalogu głównego).

Uruchom interpreter języka Python 3 raz jeszcze.

Spróbuj zaimportować swój moduł....

...ale tym razem pojawia się komunikat błędu!

Wiersz polecenia

```
C:\Users\lsuma\Desktop\Python\mymodules>cd \

C:\>py -3
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'vsearch'
>>> quit()

C:\>
```

Plik *vsearch.py* nie znajduje się już w bieżącym katalogu roboczym, ponieważ w tej chwili pracujemy w innym folderze niż *mymodules*. Oznacza to, że pliku modułu nie da się zaimportować, ponieważ nie można go znaleźć. To właśnie stąd wziął się widoczny na ekranie błąd `ModuleNotFoundErrror`.

Jeśli spróbujemy wykonać to samo ćwiczenie na platformie innej niż *Windows*, otrzymamy dokładnie taki sam wynik (niezależnie od tego, czy będziemy korzystać z systemu *Linux*, *Unix* czy *Mac OS X*). Poniżej znajdziesz przykład wcześniejszej interakcji z interpreterem przeprowadzonej z wnętrza folderu *mymodules* na platformie *OS X*.

Przejdź do folderu o nazwie „mymodules", a następnie uruchom interpreter języka Python 3, wpisując polecenie „python3".

Zaimportuj swój moduł.

Importowanie się udało, ponieważ możemy korzystać z funkcji należących do modułu.

~/Desktop/mymodules — -bash — 88×12

```
Desktop$ cd mymodules
mymodules$ python3
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('mierni')
{'e', 'i'}
>>> vsearch.search4letters('galaktyka', 'tym')
{'t', 'y'}
>>> quit()
mymodules$
```

Opuść interpreter Pythona i wróć do wiersza poleceń swojego systemu operacyjnego.

Błąd importowania pojawia się niezależnie od platformy

moduł

Jeśli myślisz, że korzystanie z platformy innej niż *Windows* w jakiś sposób pozwoli Ci uniknąć tego problemu z importowaniem modułu, z którym mieliśmy do czynienia przed chwilą, zastanów się raz jeszcze. Ten sam błąd `ModuleNotFoundError` zdarza się w systemach uniksowych, gdy tylko zmienimy bieżący katalog roboczy, co zilustrowano poniżej.

Przejdź do innego folderu (tutaj przechodzimy do katalogu najwyższego poziomu).

Ponownie uruchom interpreter języka Python 3.

Spróbuj zaimportować swój moduł....

...ale tym razem pojawia się komunikat błędu!

```
~ — -bash — 88×12
mymodules$ cd
~$ python3
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'vsearch'
>>> quit()
~$
```

Podobnie jak miało to miejsce w przypadku systemu *Windows*, również tutaj plik *vsearch.py* nie znajduje się w bieżącym katalogu roboczym interpretera, ponieważ przed uruchomieniem opuściliśmy folder *mymodules*. Oznacza to, że plik naszego modułu nie może zostać zaimportowany, ponieważ nie da się go znaleźć i właśnie to jest przyczyną wyświetlenia na ekranie przez interpreter błędu `ModuleNotFoundErrror`. Jak widać, problem ten występuje niezależnie od platformy, na której korzysta się ze środowiska Python.

Nie istnieją głupie pytania

P: Czy nie możemy bardziej precyzyjnie określić lokalizacji modułu i napisać coś w rodzaju `import C:\mymodules\vsearch`, korzystając z platformy Windows, lub też może `import /mymodules/vsearch` — w systemach uniksowych?

O: Nie, nie możemy. Z pewnością rozwiązanie to wydaje się kuszącą perspektywą, ale i tak nie zadziała, ponieważ nie można w ten sposób używać ścieżek dostępu w ramach zapewnianej przez język Python instrukcji `import`. Zresztą ostatnią rzeczą, jaką chciałbyś robić, byłoby umieszczanie w swoich programach zakodowanych na sztywno ścieżek dostępu, ponieważ mogą się one często zmieniać (z mnóstwa różnych powodów). Jeśli to tylko możliwe, zawsze najlepiej unikać jakichkolwiek zakodowanych na sztywno ścieżek dostępu w swoim kodzie.

P: Skoro nie mogę korzystać ze ścieżek dostępu, to jak mogę sprawić, aby interpreter znalazł moje moduły?

O: Gdy interpreter nie może znaleźć Twojego modułu w bieżącym katalogu roboczym, sprawdza lokalizacje **pakietów**, jak również biblioteki standardowej (więcej informacji na temat pakietów znajdziesz na następnej stronie). Jeśli możesz dodać swój moduł do jednej z lokalizacji **pakietów**, interpreter będzie w stanie go tam znaleźć (bez względu na rzeczywistą ścieżkę dostępu do pliku).

Włączanie modułu do pakietów

moduł

Przypomnij sobie, co mieliśmy Ci do przekazania na temat **pakietów** kilka stron temu, gdy przedstawiliśmy je jako drugą z trzech lokalizacji przeszukiwanych przez mechanizm importowania interpretera.

2. **Lokalizacje pakietów interpretera.**
   Są to katalogi zawierające wszystkie pochodzące od osób trzecich moduły Pythona, które mogłeś zainstalować (w tym wszystkie te, które sam napisałeś).

Jako że dostarczanie i obsługa modułów pochodzących od osób trzecich stanowi główny element strategii ponownego użycia kodu w języku Python, nie powinno Cię dziwić, że interpreter zapewnia wbudowaną możliwość dodawania modułów do Twojej konfiguracji Pythona.

Zwróć uwagę na to, że zestawem modułów należących do biblioteki standardowej zarządzają programiści odpowiedzialni za rdzeń Pythona, a ta ogromna kolekcja modułów ma mieć bardzo szerokie zastosowanie, lecz nie powinna być zmieniana przez osoby do tego niepowołane. W szczególności nie powinieneś dodawać do niej własnych modułów ani niczego z niej usuwać. Jednak dodawanie i usuwanie modułów w obrębie lokalizacji pakietów jest jak najbardziej możliwe, a nawet wskazane, dlatego Python został wyposażony w pewne narzędzia, które upraszczają ten proces.

Używanie narzędzia setuptools do instalowania w pakietach

Wraz z wydaniem wersji 3.4 języka Python w bibliotece standardowej pojawił się moduł o nazwie `setuptools`, z którego można korzystać, aby dodawać dowolne moduły do pakietów. Szczegóły dystrybucji modułów mogą się początkowo wydawać dość skomplikowane, ale nas interesuje tu na razie wyłącznie zainstalowanie modułu `vsearch` w pakietach, co za pomocą narzędzia `setuptools` można zrobić w trzech prostych krokach:

1. **Utwórz opis dystrybucji.**
   Identyfikuje on moduł, który chcemy zainstalować za pomocą narzędzia `setuptools`.

2. **Wygeneruj plik dystrybucji.**
   Korzystając z Pythona w wierszu poleceń, utworzymy plik dystrybucji zawierający kod naszego modułu, który to plik będzie można udostępniać.

3. **Zainstaluj plik dystrybucji.**
   Ponownie używając Pythona w wierszu poleceń, zainstalujemy (zawierający nasz moduł) plik dystrybucji w pakietach.

Python 3.4 (i jego nowsze wersje) znacznie ułatwia korzystanie z narzędzia setuptools. Jeśli używasz starszej wersji Pythona, rozważ jego aktualizację.

Krok 1. wymaga utworzenia (co najmniej) dwóch plików z opisem naszego modułu. Są to pliki: *setup.py* oraz *README.txt*. Przekonajmy się, co się z tym wiąże.

Tworzenie wymaganych plików konfiguracji

- [ ] Utwórz opis dystrybucji.
- [ ] Wygeneruj plik dystrybucji.
- [ ] Zainstaluj plik dystrybucji.

Będziemy zaznaczać każdy wykonany krok wraz z postępem naszej pracy.

Gdy wykonamy trzy kroki przedstawione na końcu poprzedniej strony, doprowadzimy do utworzenia **pakietu dystrybucyjnego** naszego modułu. Pakiet ten stanowi pojedynczy plik skompresowany, który zawiera wszystko, co niezbędne, aby zainstalować moduł w pakietach.

Aby wykonać krok 1., *Utwórz opis dystrybucji*, musimy powołać do życia dwa pliki, które umieścimy w tym samym folderze co nasz plik *vsearch.py*. Zrobimy to niezależnie od tego, z jakiej platformy przyszło nam korzystać. Pierwszy z plików, noszący nazwę *setup.py*, ma dość szczegółowo opisywać nasz moduł.

Poniżej znajdziesz plik *setup.py*, który utworzyliśmy, aby opisać moduł znajdujący się pliku *vsearch.py*. Zawiera on dwa wiersze kodu Pythona: w pierwszym z nich importowana jest funkcja `setup` należąca do modułu `setuptools`, w drugim zaś funkcja ta jest wywoływana.

Funkcja `setup` przyjmuje bardzo dużą liczbę argumentów, z których wiele jest opcjonalnych. Zwróć uwagę na to, jak — w celu zwiększenia czytelności kodu — nasze wywołanie funkcji `setup` zostało rozłożone na dziewięć wierszy. Korzystamy tu z zapewnianej przez Pythona obsługi argumentów ze słowami kluczowymi, aby jednoznacznie wskazać, która wartość przypisana jest któremu argumentowi. Najważniejsze argumenty zostały wyróżnione: pierwszy z nich stanowi nazwę dystrybucji, a drugi zawiera listę plików o rozszerzeniu *py*, które mają zostać uwzględnione przy tworzeniu pakietu dystrybucyjnego.

Zaimportuj funkcję setup z modułu setuptools.

```
from setuptools import setup

setup(
    name='vsearch',
    version='1.0',
    description='Narzędzia wyszukiwania dla książki Python. Rusz głową!',
    author='Python. Rusz głową! Wydanie II',
    author_email='hfpy2e@gmail.com',
    url='headfirstlabs.com',
    py_modules=['vsearch'],
)
```

Argument name identyfikuje dystrybucję. Powszechną praktyką jest nadawanie dystrybucji nazwy zgodnej z nazwą modułu.

To jest wywołanie funkcji setup. Rozłożyliśmy jej argumenty na wiele wierszy kodu.

To jest lista plików o rozszerzeniu py, które mają być włączone do pakietu. W tym przykładzie na liście tej znajduje się tylko jeden plik: vsearch.py.

Oprócz pliku *setup.py* mechanizm `setuptools` wymaga istnienia jeszcze jednego pliku, w którym można umieścić tekstowy opis swojego pakietu. Choć obecność tego pliku jest konieczna, sama jego zawartość jest opcjonalna, dlatego (na razie) możesz utworzyć pusty plik o nazwie *README.txt*, który powinien się znaleźć w tym samym folderze, w jakim został umieszczony plik *setup.py*. Wystarczy to, aby spełnić wymagania dotyczące drugiego pliku, o którym był mowa w kroku 1.

Tworzenie pliku dystrybucji

- [x] Utwórz opis dystrybucji.
- [ ] Wygeneruj plik dystrybucji.
- [ ] Zainstaluj plik dystrybucji.

Na tym etapie powinieneś mieć trzy pliki, które znajdują się w naszym folderze o nazwie *mymodules*. Są to: *vsearch.py*, *setup.py* oraz *README.txt*.

Teraz jesteśmy gotowi, aby z tych trzech plików utworzyć pakiet dystrybucyjny. Jest to krok 2. naszej wcześniejszej listy: *Wygeneruj plik dystrybucji*. Zrobimy to z poziomu systemowego wiersza poleceń. Choć sama operacja jest dość prosta, trzeba tu wprowadzić różne polecenia w zależności od tego, czy używa się systemu *Windows*, czy też jednego z uniksowych systemów operacyjnych (a więc systemu *Linux*, *Unix* lub *Mac OS X*).

Tworzenie pliku dystrybucji w systemie Windows

Jeśli korzystasz z systemu *Windows*, otwórz wiersz poleceń w folderze zawierającym Twoje trzy pliki, a następnie wprowadź polecenie przedstawione poniżej.

Uruchom interpreter Pythona 3 w systemie Windows.

```
c:\Users\lsuma\Desktop\Python\mymodules> py -3 setup.py sdist
```

Wykonaj kod programu setup.py...

...i przekaż sdist jako argument.

Interpreter rusza do pracy natychmiast po tym, gdy zatwierdzisz to polecenie. Na ekranie pojawi się spora liczba komunikatów, które zostały pokazane poniżej w nieco skróconej formie.

```
running sdist
running egg_info
creating vsearch.egg-info
...
creating vsearch-1.0
creating vsearch-1.0\vsearch.egg-info
copying files to vsearch-1.0...
copying README.txt -> vsearch-1.0
...
creating dist
Creating tar archive
removing 'vsearch-1.0' (and everything under it)
```

Jeśli zobaczysz ten komunikat, to znaczy, że wszystko poszło dobrze. Jeśli ujrzysz jakieś komunikaty błędów, sprawdź, czy korzystasz z Pythona w wersji 3.4 lub nowszej, a także upewnij się, że Twój plik setup.py jest dokładnie taki sam jak nasz.

Gdy ponownie pojawi się znak zachęty wiersza poleceń systemu *Windows*, Twoje trzy pliki powinny już być połączone w jeden **plik dystrybucji**. Jest to plik, który można zainstalować i który zawiera kod źródłowy Twojego modułu. W naszym przypadku plik ten nosi nazwę *vsearch-1.0.tar.gz*.

Nowo utworzone archiwum znajdziesz w folderze o nazwie *dist*, który również został utworzony przez narzędzie `setuptools` w folderze, z którego korzystasz (w naszym przypadku jest nim folder *mymodules*).

Pliki dystrybucji w systemach uniksowych

- [x] Utwórz opis dystrybucji.
- [ ] Wygeneruj plik dystrybucji.
- [ ] Zainstaluj plik dystrybucji.

Jeśli nie używasz systemu *Windows*, możesz utworzyć plik dystrybucji w sposób bardzo podobny do tego, który został zaprezentowany na poprzedniej stronie. Mając swoje trzy pliki (*vsearch.py*, *setup.py* oraz *README.txt*) w folderze, wpisz i zatwierdź przedstawione poniżej polecenie w systemowym wierszu poleceń.

Uruchom interpreter Pythona 3.

```
mymodules$ python3 setup.py sdist
```

Wykonaj kod programu setup.py...

...i przekaż sdist jako argument.

Podobnie jak w przypadku systemu *Windows*, spowoduje to wyświetlenie na ekranie wielu komunikatów, z których wybrane pokazane zostały poniżej.

```
running sdist
running egg_info
creating vsearch.egg-info
...
running check
creating vsearch-1.0
creating vsearch-1.0/vsearch.egg-info
...
creating dist
Creating tar archive
removing 'vsearch-1.0' (and everything under it)
```

Komunikaty mogą się tu nieznacznie różnić od tych, które można było zobaczyć w systemie Windows. Jeśli jednak zobaczysz ten komunikat, to znaczy, że wszystko poszło dobrze. Jeśli nie, sprawdź wszystko raz jeszcze (podobnie jak miało to miejsce w przypadku Windowsa).

Gdy ponownie pojawi się znak zachęty systemowego wiersza poleceń, Twoje trzy pliki powinny już być połączone w plik **źródła dystrybucji** (ang. *source distribution*; stąd właśnie wziął się argument `sdist`). Jest to plik, który można zainstalować i który zawiera kod źródłowy Twojego modułu. W naszym przypadku plik ten nosi nazwę *vsearch-1.0.tar.gz*.

Nowo utworzone archiwum znajdziesz w folderze o nazwie *dist*, który również został utworzony przez narzędzie `setuptools` w folderze, z którego korzystasz (w naszym przypadku jest nim folder *mymodules*).

Skoro udało Ci się już utworzyć plik źródła dystrybucji (w postaci skompresowanego archiwum tar), jesteś gotów zainstalować swój moduł w pakietach.

Instalowanie pakietów za pomocą narzędzia pip

- [x] Utwórz opis dystrybucji.
- [x] Wygeneruj plik dystrybucji.
- [ ] Zainstaluj plik dystrybucji.

Teraz, gdy dysponujesz już swoim plikiem dystrybucji w postaci skompresowanego archiwum, przyszedł czas na wykonanie kroku 3.: *Zainstaluj plik dystrybucji*. Python upraszcza, jak wiele innych kwestii, również tę czynność, zapewniając odpowiednie narzędzie. W przypadku Pythona w wersji 3.4 (oraz nowszych) jest nim narzędzie `pip`, którego nazwa to skrót utworzony z angielskich słów ***P**ackage **I**nstaller for **P**ython* (czyli instalator pakietów dla Pythona).

Krok 3. w systemie Windows

Znajdź swój nowo utworzony plik archiwum w folderze *dist* (przypomnij sobie, że plik nosi nazwę *vsearch-1.0.tar.gz*). Korzystając z okna *Eksplorator plików*, przytrzymaj klawisz *Shift*, a następnie wywołaj menu kontekstowe za pomocą prawego przycisku myszy. Wybierz z niego polecenie *Otwórz okno polecenia tutaj*. Na ekranie pojawi się okno wiersza poleceń systemu *Windows*. Używając go, wprowadź przedstawione poniżej polecenie, aby wykonać krok 3.

Uruchom Pythona 3 z modułem pip, po czym poproś ten moduł o zainstalowanie podanego pliku archiwum.

```
c:\Users\...\dist> py -3 -m pip install vsearch-1.0.tar.gz
```

Jeśli próba wykonania tego polecenia zakończy się niepowodzeniem i zgłoszony zostanie błąd uprawnień, będziesz prawdopodobnie musiał ponownie uruchomić wiersz poleceń jako administrator systemu *Windows*, a następnie spróbować wykonać to polecenie raz jeszcze.

Jeśli wykonanie przedstawionego powyżej polecenia się powiedzie, na ekranie pojawią się widoczne poniżej komunikaty.

```
Processing c:\users\...\dist\vsearch-1.0.tar.gz
Installing collected packages: vsearch
  Running setup.py install for vsearch ... done
Successfully installed vsearch-1.0
```

Sukces!

Krok 3. w systemach uniksowych

W systemie *Linux*, *Unix* lub *Mac OS X* otwórz terminal w nowo utworzonym folderze *dist*, a następnie wprowadź i zatwierdź polecenie, które zostało pokazane poniżej.

Uruchom Pythona 3 z modułem pip, po czym poproś ten moduł o zainstalowanie podanego pliku archiwum.

```
.../dist$ sudo python3 -m pip install vsearch-1.0.tar.gz
```

Używamy tu polecenia sudo, aby mieć pewność, że instalujemy nasz plik z odpowiednimi uprawnieniami.

Jeśli wykonanie przedstawionego powyżej polecenia się powiedzie, na ekranie pojawią się pokazane tutaj komunikaty.

```
Processing ./vsearch-1.0.tar.gz
Installing collected packages: vsearch
  Running setup.py install for vsearch ... done
Successfully installed vsearch-1.0
```

Sukces!

Moduł vsearch jest teraz zainstalowany jako część pakietów.

Moduły: co już wiemy

- ✔ Utwórz opis dystrybucji.
- ✔ Wygeneruj plik dystrybucji.
- ✔ Zainstaluj plik dystrybucji.

Wszystko zrobione!

Po zainstalowaniu naszego modułu `vsearch` możemy używać instrukcji `import vsearch` w każdym z naszych programów, mając pewność, że interpreter jest w stanie znaleźć należące do modułu funkcje, których potrzebujemy.

Jeśli zdecydujemy się w jakimś momencie w przyszłości zmodyfikować kod naszego modułu, będziemy mogli powtórzyć te trzy kroki, aby zainstalować wszystkie aktualizacje w pakietach. Jeśli utworzysz nową wersję swojego modułu, pamiętaj, aby przypisać jej nowy numer wersji w pliku *setup.py*.

CELNE SPOSTRZEŻENIA

- Moduł to jedna funkcja lub większa ich liczba zapisana w pliku.
- Moduł można udostępniać, zapewniając, że zawsze znajduje się w *bieżącym katalogu roboczym* interpretera (co da się zrobić, ale co nie jest dobrym rozwiązaniem) albo w obrębie jego *lokalizacji pakietów* (co jest znacznie lepszym pomysłem).
- Wykonanie trzech kroków procedury związanej z użyciem narzędzia `setuptools` zapewnia, że moduł zostaje zainstalowany w pakietach, dzięki czemu można go importować za pomocą instrukcji `import` i korzystać z należących do niego funkcji niezależnie od tego, jaki jest w danej chwili *bieżący katalog roboczy*.

Przekazywanie kodu dalej (znane również jako udostępnianie)

Gdy masz już utworzony plik dystrybucji, możesz go udostępnić innym programistom Pythona, umożliwiając również im instalację Twojego modułu za pomocą narzędzia `pip`. Swój plik możesz udostępniać w jeden z dwóch sposobów: nieformalnie lub formalnie.

Aby udostępnić swój moduł nieformalnie, po prostu rozdystrybuuj go w dowolny sposób, który przyjdzie Ci do głowy, i wśród wszystkich tych osób, którym będziesz go chciał przekazać (na przykład wysyłając plik pocztą elektroniczną, zapisując w pamięci przenośnej USB lub udostępniając do ściągnięcia na Twojej osobistej stronie internetowej). Naprawdę możesz to zrobić, jeśli tylko zechcesz.

Aby udostępnić swój moduł formalnie, możesz wrzucić plik dystrybucji do centralnie zarządzanego, dostępnego poprzez sieć repozytorium kodu, które nosi nazwę PyPI (co jest skrótem od angielskich słów *Python Package Index*, oznaczających indeks pakietów Pythona). Celem istnienia tego serwisu jest umożliwienie wszystkim programistom Pythona udostępnianie wszelkiej maści modułów pochodzących od osób trzecich. Więcej informacji na temat tego, co PyPI ma do zaoferowania, znajdziesz pod adresem ***https://pypi.python.org/pypi***. Dodatkowe dane o procedurze wrzucania i udostępniania plików dystrybucji za pośrednictwem PyPI opublikowano w dostępnym online przewodniku opracowanym przez *Python Packaging Authority*, który można znaleźć pod adresem ***https://www.pypa.io***. (W gruncie rzeczy nie ma tu wiele więcej do powiedzenia, ale i te szczegóły wykraczają poza materiał prezentowany w tej książce).

Każdy programista Pythona również może korzystać z narzędzia pip, aby zainstalować Twój moduł.

Dotarliśmy już niemal do końca naszego wprowadzenia do funkcji i modułów. Jest jeszcze tylko jedna mała zagadka, która wymaga naszej uwagi (przez nie więcej niż pięć minut). Jeśli będziesz gotów, przejdź do następnej strony.

Przypadek źle zachowujących się argumentów funkcji

Tomek i Sara właśnie przeczytali ten rozdział i kłócą się teraz o sposób, w jaki ich zdaniem zachowują się argumenty funkcji w Pythonie.

Tomek jest przekonany, że gdy argumenty są przesyłane do funkcji, dane przekazywane są **przez wartość**. Aby udowodnić swoje twierdzenie, napisał nawet niewielką funkcję o nazwie `double`. Funkcja ta operuje na danych dowolnego typu, które zostaną jej przekazane.

Zagadka na pięć minut

Oto kod opracowany przez Tomka:

```
def double(arg):
    print('Przed: ', arg)
    arg = arg * 2
    print('Po:    ', arg)
```

Sara jest z kolei pewna, że gdy argumenty są przesyłane do funkcji, dane przekazywane są **przez referencje**. Również ona napisała krótką funkcję, noszącą nazwę `change`, która operuje na listach i ma potwierdzić jej stanowisko.

Oto kod Sary:

```
def change(arg):
    print('Przed: ', arg)
    arg.append('Więcej danych')
    print('Po:    ', arg)
```

Wolelibyśmy, aby nikt nie kłócił się z takiego powodu, zwłaszcza że — jak dotąd — Tomek i Sara stanowili zgraną paczkę programistów. Poeksperymentujmy w wierszu poleceń powłoki, aby przekonać się, kto ma rację: Tomek „przez wartość” czy Sara „przez referencję”. Przecież nie mogą mieć jej oboje, prawda? Zdecydowanie jest to zagadka, którą należy rozwikłać, a to prowadzi do często zadawanego pytania:

Czy argumenty funkcji w Pythonie używają semantyki „przez wartość”, czy „przez referencję”?

Porady maniaka

Gdybyś potrzebował szybkiego odświeżenia pamięci, przypomnij sobie, że **przekazywanie argumentu przez wartość** oznacza praktykę polegającą na używaniu wartości przechowywanej w zmiennej w miejscu argumentu funkcji. Jeśli wartość ta ulegnie zmianie w zestawie funkcji, nie ma to wpływu na wartość zmiennej w kodzie wywołującym funkcję. O argumencie możesz w takim przypadku myśleć jako o *kopii* oryginalnej wartości zmiennej. **Przekazywanie argumentu przez referencję** (określane też czasem jako **przekazywanie argumentu przez adres**) wiąże się z używaniem odwołania do zmiennej w kodzie wywołującym funkcję. Jeśli wartość tej zmiennej ulegnie zmianie w zestawie funkcji, jej wartość w kodzie wywołującym również zostanie zmieniona. O argumencie możesz w takim przypadku myśleć jako o *synonimie* (czy też *aliasie*) oryginalnej zmiennej.

Prezentacja semantyki „wywołania przez wartość”

Aby przekonać się, o co tak naprawdę kłócą się Tomek i Sara, umieśćmy ich funkcje w osobnym module, który nazwiemy *mystery.py*. Tak oto kod tego modułu prezentuje się w oknie edycji środowiska IDLE.

Te dwie funkcje są do siebie podobne. Każda przyjmuje pojedynczy argument, wyświetla go na ekranie, zmienia jego wartość, a następnie raz jeszcze wyświetla go na ekranie.

mystery.py - C:/Users/lsuma/Desktop/Python/R04/mystery.py...

File Edit Format Run Options Window Help

```
def double(arg):
    print('Przed: ', arg)
    arg = arg * 2
    print('Po:    ', arg)

def change(arg):
    print('Przed: ', arg)
    arg.append('Więcej danych')
    print('Po:    ', arg)
```

Ln: 11 Col: 0

Ta funkcja podwaja przekazaną wartość.

Ta funkcja dołącza łańcuch znakowy do dowolnej listy, która zostanie jej przekazana.

Tomek, gdy tylko ujrzał ten moduł na ekranie, natychmiast usiadł do klawiatury, nacisnął klawisz *F5* i w wierszu poleceń powłoki wykonał operacje przedstawione poniżej. Po skończeniu rozsiadł się wygodnie w swoim fotelu, skrzyżował ramiona i stwierdził: „Widzisz? Mówiłem ci, że to wywołanie przez wartość”. Przyjrzyj się działaniom Tomka w powłoce.

```
>>> num = 10
>>> double(num)
Przed:  10
Po:     20
>>> num
10
>>> saying = 'Halo '
>>> double(saying)
Przed:  Halo
Po:     Halo Halo
>>> saying
'Halo '
>>> numbers = [ 42, 256, 16 ]
>>> double(numbers)
Przed:  [42, 256, 16]
Po:     [42, 256, 16, 42, 256, 16]
>>> numbers
[42, 256, 16]
```

Tomek wywołuje funkcję double trzy razy: raz z wartością całkowitą, następnie z łańcuchem znakowym, a na koniec z listą.

Każde wywołanie potwierdza, że wartość przekazana jako argument zostaje zmodyfikowana w zestawie funkcji, ale wartość w powłoce pozostaje niezmienna. A zatem wygląda na to, że mamy tu do czynienia z semantyką wywołania przez wartość.

Prezentacja semantyki „wywołania przez referencję"

Sara niezrażona pozornym zwycięstwem Tomka usiadła do komputera i przejęła kontrolę nad klawiaturą, aby przeprowadzić swoje próby w powłoce. Poniżej raz jeszcze przedstawiliśmy kod naszego modułu w oknie edycji środowiska IDLE, a w nim — funkcję Sary gotową do działania.

Moduł mystery.py

mystery.py - C:/Users/lsuma/Desktop/Python/R04/mystery.py...

File Edit Format Run Options Window Help

```
def double(arg):
    print('Przed: ', arg)
    arg = arg * 2
    print('Po:    ', arg)

def change(arg):
    print('Przed: ', arg)
    arg.append('Więcej danych')
    print('Po:    ', arg)
```

Funkcja Tomka

Funkcja Sary

Ln: 11 Col: 0

Sara wprowadziła kilka wierszy kodu po znaku zachęty >>> wiersza poleceń powłoki Pythona, po czym rozsiadła się w fotelu i powiedziała do Tomka: „Cóż, jeśli Python korzysta wyłącznie z wywołań przez wartość, jak wytłumaczysz to, co się tu dzieje?". Tomek zaniemówił.

Przyjrzyj się pokazanym poniżej operacjom wykonanym przez Sarę w wierszu poleceń powłoki.

Korzystając z tej samej listy, co Tomek, Sara wywołuje swoją funkcję change.

```
>>> numbers = [ 42, 256, 16 ]
>>> change(numbers)
Przed:  [42, 256, 16]
Po:     [42, 256, 16, 'Więcej danych']
>>> numbers
[42, 256, 16, 'Więcej danych']
```

Zobacz, co się stało! Tym razem wartość argumentu została zmieniona zarówno w funkcji, jak i w powłoce. Wydaje się to sugerować, że funkcje Pythona korzystają RÓWNIEŻ z semantyki wywołania przez referencję.

To zachowanie *jest* dziwne.

Sposób działania funkcji Tomka jasno wskazuje, że mamy do czynienia z semantyką wywołania przez wartość, podczas gdy przykład zastosowania funkcji Sary wyraźnie demonstruje wywołanie przez referencję.

Jak to możliwe? Co się tu dzieje? Czy język Python korzysta z *obydwu* tych sposobów przekazywania argumentów?

Przypadek źle zachowujących się argumentów funkcji — rozwiązanie

Czy argumenty funkcji Pythona korzystają z semantyki wywołania przez wartość, czy przez referencję?

A oto niespodzianka: zarówno Tomek, *jak i* Sara mają rację. W zależności od sytuacji argumenty funkcji Pythona mogą wykorzystywać **zarówno** semantykę wywołania przez wartość, *jak i* semantykę wywołania przez referencję.

Przypomnij sobie raz jeszcze, że zmienne w Pythonie są w rzeczywistości **odwołaniami do obiektów**. O wartości przechowywanej w zmiennej warto tutaj myśleć jako o adresie w pamięci, pod którym się ona znajduje, nie zaś jako o samej wartości. Oznacza to, że funkcje Pythona korzystają w istocie z czegoś, co bardziej poprawnie nazywane jest *semantyką wywołania przez referencję obiektu*.

Rzeczywista semantyka wywołania używana w danej chwili może się różnić w zależności od typu obiektu, do którego się odwołujemy. Jak to zatem możliwe, że w przypadku funkcji Tomka i Sary argumenty wpisywały się w obydwie semantyki naraz? Przede wszystkim wcale się nie wpisywały w nie, a jedynie zdawały się wpisywać. Tak naprawdę interpreter sprawdza tu typ wartości, do której następuje odwołanie przez referencję do obiektu (adres w pamięci), i jeśli odwołanie to dotyczy wartości **modyfikowalnej**, stosuje semantykę wywołania przez referencję. Jeśli zaś typ danej, do której się odwołujemy, jest **niemodyfikowalny**, do gry wchodzi semantyka wywołania przez wartość. Zastanów się teraz, co oznacza to w przypadku naszych danych.

Listy, słowniki i zbiory (będące strukturami modyfikowalnymi) są zawsze przekazywane funkcji przez referencję, a więc wszelkie zmiany dokonywane w strukturze danych reprezentowanej przez zmienną w zestawie funkcji są odzwierciedlane w kodzie wywołującym. Dane są w końcu modyfikowalne.

Łańcuchy znakowe, wartości całkowite i krotki (będące niemodyfikowalnymi danymi) są zawsze przekazywane funkcji przez wartość, a więc wszelkie zmiany dokonywane w obrębie funkcji mają dla niej prywatny charakter i nie są odzwierciedlane w kodzie wywołującym. Dzieje się tak, ponieważ dane niemodyfikowalne nie mogą ulec zmianie.

Zagadka na pięć minut

Wszystko to wydaje się mieć sens, dopóki nie przyjrzysz się dokładniej temu wierszowi kodu:

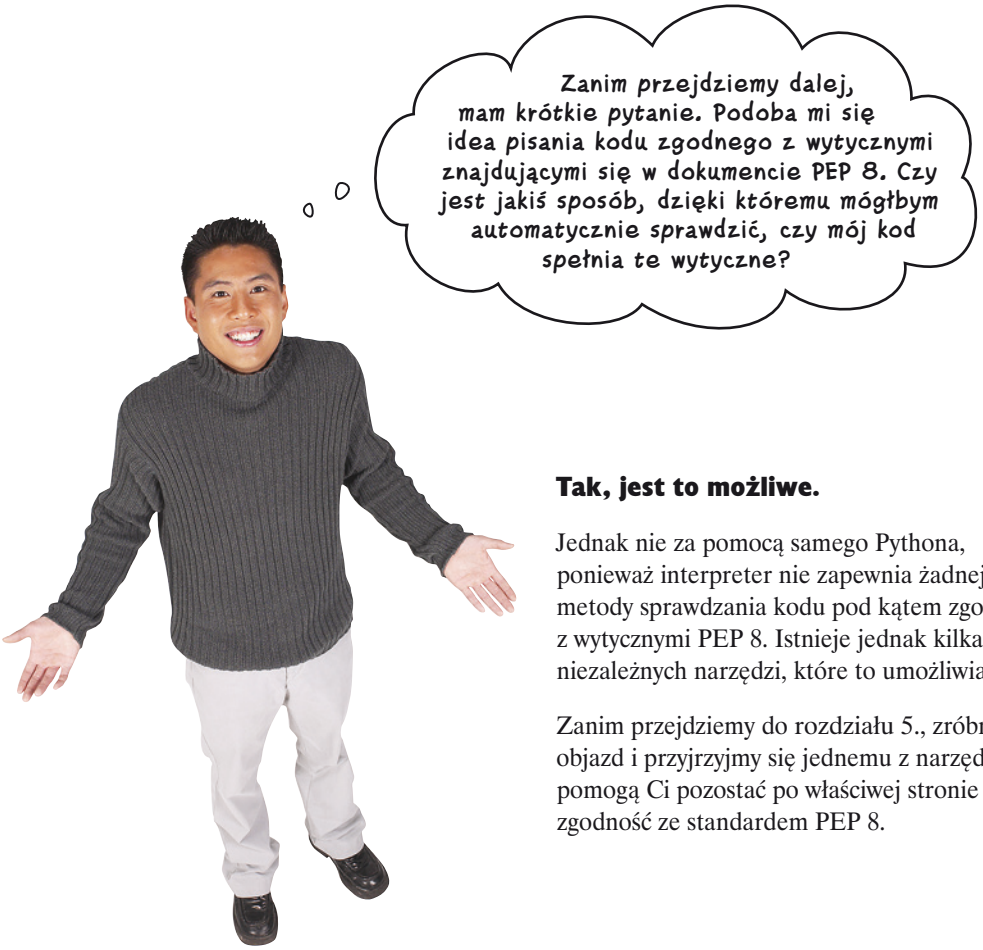
```
arg = arg * 2
```

Jak to możliwe, że wiersz ten najwyraźniej powodował zmianę przekazanej listy w obrębie zestawu funkcji, lecz gdy lista ta została wyświetlona w powłoce już po wywołaniu funkcji, pozostała niezmieniona (co doprowadziło Tomka do — nieprawidłowego — wniosku, że wszystkie argumenty przekazywane były przez wartość)? Z pozoru może się to wydawać bugiem w interpreterze, bo przecież stwierdziliśmy już, że zmiany dokonywane w stosunku do wartości modyfikowalnych są odzwierciedlane w kodzie wywołującym, a tu tak się nie dzieje. A zatem funkcja Tomka *nie* zmieniła listy `numbers` w kodzie wywołującym, choć listy są modyfikowalne. Co więc jest tu grane?

Zauważ, że powyższy wiersz kodu stanowi **instrukcję przypisania**. Oto co się dzieje w trakcie przypisywania: kod znajdujący się z prawej strony symbolu = jest wykonywany *najpierw*, a następnie referencja do obiektu dowolnej wartości, która w wyniku tego powstaje, zostaje przypisana do zmiennej widocznej po lewej stronie symbolu =. Wykonanie kodu `arg * 2` skutkuje powstaniem *nowej* wartości, której przypisywana jest *nowa* referencja do obiektu, która jest następnie przypisywana do zmiennej `arg`, co powoduje zastąpienie nią poprzedniej referencji do obiektu przechowywanej w zmiennej `arg` w zestawie funkcji. Jednakże „stara” referencja do obiektu nadal istnieje w kodzie wywołującym, a jej wartość nie została zmieniona, dlatego powłoka w dalszym ciągu „widzi” oryginalną listę, a nie nową, podwojoną listę utworzoną za pomocą kodu Tomka. Porównaj to zachowanie do działania kodu Sary, w którym wywoływana jest metoda `append` na rzecz istniejącej listy. Jako że nie mamy tu do czynienia z żadną operacją przypisania, nie odbywa się też żadne nadpisanie referencji do obiektu, dlatego kod Sary modyfikuje również listę powłoki. Jest tak, ponieważ zarówno lista, do której odwołujemy się w zestawie funkcji, jak i lista, do której odwołuje się kod wywołujący, przechowują *tę samą* referencję do obiektu.

Rozwiązaliśmy tę zagadkę. Została nam do omówienia już tylko jedna istotna kwestia.

Czy mogę sprawdzić zgodność ze standardem PEP 8?



Tak, jest to możliwe.

Jednak nie za pomocą samego Pythona, ponieważ interpreter nie zapewnia żadnej metody sprawdzania kodu pod kątem zgodności z wytycznymi PEP 8. Istnieje jednak kilka niezależnych narzędzi, które to umożliwiają.

Zanim przejdziemy do rozdziału 5., zróbmy mały objazd i przyjrzyjmy się jednemu z narzędzi, które pomogą Ci pozostać po właściwej stronie i zapewnić zgodność ze standardem PEP 8.

Przygotowanie do sprawdzania zgodności ze standardem PEP 8

Odejdźmy na moment od głównego tematu, aby sprawdzić, czy nasz kod jest zgodny z wytycznymi PEP 8

Duża część społeczności programistów Pythona poświęciła sporo czasu na opracowanie narzędzi programistycznych, które choć trochę ułatwiają życie użytkownikom języka. Jednym z takich narzędzi jest `pytest`, będący *frameworkiem do testowania*, którego głównym zadaniem jest uproszczenie przeprowadzania testów programów napisanych w Pythonie. Niezależnie od tego, jakiego rodzaju testy masz zamiar pisać, pomoże Ci w tym `pytest`. Do narzędzia `pytest` możesz ponadto dodawać wtyczki, które zwiększają jego możliwości.

Jedną z takich wtyczek jest `pep8`, która wykorzystuje framework testowy `pytest` do sprawdzania, czy kod narusza standardy zdefiniowane przez wytyczne PEP 8.

Więcej informacji na temat narzędzia pytest znajdziesz pod adresem *http://doc.pytest.org/en/latest/*.

Przypomnienie naszego kodu

Raz jeszcze przypomnijmy sobie kod znajdujący się w pliku *vsearch.py*, zanim przekażemy go „w ręce” zespołu składającego się z narzędzi **pytest** i **pep8** w celu sprawdzenia, jak bardzo zgodny jest z wymaganiami stawianymi przez standard PEP 8. Zwróć uwagę na to, że musimy zainstalować obydwa te narzędzia programistyczne, ponieważ nie są dostarczane wraz z Pythonem (sposób instalacji poznasz jeszcze na tej stronie).

```
def search4vowels(phrase:str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Zwraca zbiór liter ze zmiennej letters znalezionych w zmiennej phrase."""
    return set(letters).intersection(set(phrase))
```

Ten kod znajduje się w pliku vsearch.py.

Instalacja narzędzia pytest i wtyczki pep8

We wcześniejszej części tego rozdziału korzystałeś z narzędzia `pip`, aby zainstalować swój moduł *vsearch.py* w interpreterze Pythona działającym na Twoim komputerze. Narzędzia tego możesz również używać do instalowania kodu pochodzącego od osób trzecich.

Aby to zrobić, powinieneś przejść do wiersza poleceń swojego systemu operacyjnego (i mieć połączenie z internetem). Z narzędzia `pip` będziesz korzystał w następnym rozdziale, aby zainstalować niezależną bibliotekę. Teraz jednak użyjemy go w celu zainstalowania frameworka do testowania `pytest` oraz wtyczki `pep8`.

Instalacja narzędzi programistycznych do testowania

Na zamieszczonych tutaj rysunkach możesz zobaczyć komunikaty, które pojawiają się na ekranie podczas instalacji narzędzia `pytest` na platformie *Windows*. W tym systemie operacyjnym Pythona 3 wywołuje się za pomocą polecenia `py -3`. Jeśli używasz systemu *Linux* lub *Mac OS X*, zastąp je poleceniem `sudo python3`. Aby zainstalować framework **pytest** za pomocą narzędzia `pip` w systemie *Windows*, z poziomu systemowego wiersza poleceń wprowadź i zatwierdź przedstawione poniżej polecenie jako administrator (w tym celu poszukaj programu *cmd.exe*, kliknij go prawym przyciskiem myszy i z menu kontekstowego wybierz polecenie *Uruchom jako administrator*).

```
py -3 -m pip install pytest
```

Zacznij w trybie administratora...

...następnie wydaj polecenie pip, aby zainstalować framework pytest...

...po czym sprawdź, czy udało się go prawidłowo zainstalować.

Administrator: Wiersz polecenia

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\WINDOWS\system32>py -3 -m pip install pytest
Collecting pytest
  Downloading pytest-3.1.2-py2.py3-none-any.whl (180kB)
    100% |                                | 184kB 930kB/s
Collecting py>=1.4.33 (from pytest)
  Downloading py-1.4.34-py2.py3-none-any.whl (84kB)
    100% |                                | 92kB 1.3MB/s
Collecting colorama; sys_platform == "win32" (from pytest)
  Downloading colorama-0.3.9-py2.py3-none-any.whl
Requirement already satisfied: setuptools in c:\users\lsuma\appdata\local\programs\
python\python36\lib\site-packages (from pytest)
Installing collected packages: py, colorama, pytest
Successfully installed colorama-0.3.9 py-1.4.34 pytest-3.1.2

C:\WINDOWS\system32>
```

Gdy przejrzysz komunikaty wygenerowane przez narzędzie `pip`, zauważysz, że zostały zainstalowane również dwie zależności frameworka **pytest** (są to: **colorama** oraz **py**). To samo stanie się, gdy skorzystasz z narzędzi **pip**, aby zainstalować wtyczkę **pep8**: również ona wymaga instalacji pewnych zależności. Poniżej znajdziesz polecenie, za pomocą którego można dodać tę wtyczkę.

```
py -3 -m pip install pytest-pep8
```

Pamiętaj, że używając systemu operacyjnego innego niż Windows, powinieneś zastąpić polecenie py -3 poleceniem sudo python3.

W dalszym ciągu korzystając z trybu administratora, wydaj powtórnie polecenie instalacji wtyczki pep8.

Administrator: Wiersz polecenia

```
C:\WINDOWS\system32>py -3 -m pip install pytest-pep8
Collecting pytest-pep8
  Downloading pytest-pep8-1.0.6.tar.gz
Collecting pytest-cache (from pytest-pep8)
  Downloading pytest-cache-1.0.tar.gz
Requirement already satisfied: pytest>=2.4.2 in c:\users\lsuma\appdata\local\progra
ms\python\python36\lib\site-packages (from pytest-pep8)
Collecting pep8>=1.3 (from pytest-pep8)
  Downloading pep8-1.7.0-py2.py3-none-any.whl (41kB)
    100% |                                | 51kB 659kB/s
Collecting execnet>=1.1.dev1 (from pytest-cache->pytest-pep8)
  Downloading execnet-1.4.1-py2.py3-none-any.whl (40kB)
    100% |                                | 40kB 791kB/s
Requirement already satisfied: py>=1.4.33 in c:\users\lsuma\appdata\local\programs\
python\python36\lib\site-packages (from pytest>=2.4.2->pytest-pep8)
Requirement already satisfied: colorama; sys_platform == "win32" in c:\users\lsuma\
appdata\local\programs\python\python36\lib\site-packages (from pytest>=2.4.2->pytes
t-pep8)
Requirement already satisfied: setuptools in c:\users\lsuma\appdata\local\programs\
python\python36\lib\site-packages (from pytest>=2.4.2->pytest-pep8)
Collecting apipkg>=1.4 (from execnet>=1.1.dev1->pytest-cache->pytest-pep8)
  Downloading apipkg-1.4-py2.py3-none-any.whl
Installing collected packages: apipkg, execnet, pytest-cache, pep8, pytest-pep8
  Running setup.py install for pytest-cache ... done
  Running setup.py install for pytest-pep8 ... done
Successfully installed apipkg-1.4 execnet-1.4.1 pep8-1.7.0 pytest-cache-1.0 pytest-
pep8-1.0.6

C:\WINDOWS\system32>
```

Również to polecenie zostało wykonane prawidłowo i także w tym przypadku zostały zainstalowane wymagane zależności.

Jak bardzo zgodny z PEP 8 jest nasz kod?

Jeśli udało Ci się już zainstalować narzędzia **pytest** oraz **pep8**, jesteś gotów do przetestowania swojego kodu pod kątem zgodności ze standardem wyznaczanym przez dokument PEP 8. Niezależnie od tego, z jakiego systemu operacyjnego korzystasz, powinieneś w tym celu wydać dokładnie to samo polecenie (ponieważ jedynie instrukcje instalacji są różne dla różnych platform).

Wykonanie procedury instalacji narzędzia **pytest** spowodowało pojawienie się na Twoim komputerze nowego programu o nazwie `py.test`. Uruchommy go, aby zbadać nasz kod zapisany w pliku *vsearch.py* pod kątem zgodności z PEP 8. Pamiętaj: zanim uruchomisz przedstawione poniżej polecenie, przejdź do folderu, w którym znajduje się plik *vsearch.py*.

```
py.test --pep8 vsearch.py
```

Poniżej możesz zobaczyć efekt wywołania tego polecenia na naszym komputerze działającym pod kontrolą systemu operacyjnego *Windows*.

Ojojoj, komunikaty oznaczone kolorem czerwonym nie mogą być dla nas dobre, prawda?

```
C:\WINDOWS\system32\cmd.exe

c:\Users\lsuma\Desktop\Python\mymodules>py.test --pep8 vsearch.py
============================= test session starts ==============================
platform win32 -- Python 3.6.0, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
rootdir: c:\Users\lsuma\Desktop\Python\mymodules, inifile:
plugins: pep8-1.0.6
collected 1 items

vsearch.py F

=================================== FAILURES ===================================
_____________________________________ PEP8-check ______________________________
c:\Users\lsuma\Desktop\Python\mymodules\vsearch.py:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                         ^
c:\Users\lsuma\Desktop\Python\mymodules\vsearch.py:7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
^
c:\Users\lsuma\Desktop\Python\mymodules\vsearch.py:7:26: E231 missing whitespace after ':'
def search4letters(phrase:str, letters:str='aeiou') -> set:
                          ^
c:\Users\lsuma\Desktop\Python\mymodules\vsearch.py:7:39: E231 missing whitespace after ':'
def search4letters(phrase:str, letters:str='aeiou') -> set:
                                       ^
c:\Users\lsuma\Desktop\Python\mymodules\vsearch.py:8:80: E501 line too long (80 > 79 characters)
    """Zwraca zbiór liter ze zmiennej letters znalezionych w zmiennej phrase."""
                                                                               ^

========================== 1 failed in 0.05 seconds ===========================

c:\Users\lsuma\Desktop\Python\mymodules>_
```

Ups! Wygląda na to, że nasz test zakończył się **niepowodzeniem**, co oznacza, że kod nie jest tak zgodny z wytycznymi PEP 8, jak powinien być.

Poświęć chwilę na przeczytanie komunikatów widocznych powyżej (lub na ekranie Twojego komputera, jeśli robisz to, co my). Wszystkie „problemy” wydają się jakoś odnosić do *białych znaków* (a więc spacji, tabulacji, znaków nowego wiersza i tym podobnych). Przyjrzyjmy się nieco dokładniej każdemu z nich.

Znaczenie komunikatów błędów

Narzędzie **pytest** wraz z wtyczką **pep8** wskazało nam w sumie *pięć* problemów występujących w naszym kodzie zapisanym w pliku *vsearch.py*.

Pierwszy z nich jest związany z tym, że nie umieściliśmy spacji po znaku `:` podczas dodawania adnotacji do argumentów funkcji. Błąd ten popełniliśmy w trzech miejscach. Popatrz na pierwszy komunikat i zwróć uwagę na sposób, w jaki narzędzie **pytest** używa znaku *daszka* (^), aby wskazać dokładne miejsce występowania problemu.

```
...:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                         ^
```

Tutaj znajduje się informacja, co jest nie tak.

Tutaj znajduje się informacja, gdzie występuje problem.

Przyglądając się pozostałym komunikatom, zauważysz, że błąd ten zrobiliśmy w trzech miejscach: raz w wierszu drugim oraz dwukrotnie w wierszu siódmym. Problem można łatwo wyeliminować: wystarczy *dodać pojedynczy znak spacji po każdym wskazanym dwukropku*.

Poprawiając wiersz siódmy, możemy od razu pozbyć się problemu wskazywanego przez kolejny komunikat.

```
...:7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
                                                     ^
```

Tutaj znajduje się informacja, co jest nie tak.

Tutaj znajduje się informacja, gdzie występuje problem.

PEP 8 zawiera wytyczną regulującą kwestie związane z tworzeniem funkcji w module, która mówi, że *funkcje najwyższego poziomu oraz definicje klas powinny być rozdzielone dwoma pustymi wierszami*. W przypadku naszego pliku *vsearch.py* funkcje `search4vowels` i `search4letters` są funkcjami „najwyższego poziomu” i są od siebie oddzielone pojedynczym pustym wierszem. Aby spełnić wymogi stawiane przez dokument PEP 8, powinniśmy tu wstawić *dwa* puste wiersze.

Również ten błąd łatwo jest naprawić: wystarczy *wstawić dodatkowy pusty wiersz pomiędzy te dwie funkcje*.

Ostatnim z wykrytych problemów jest zbyt długi wiersz tekstu dokumentującego, występującego w wierszu ósmym.

```
...:8:80: E501 line too long (80 > 79 characters)
"""Zwraca zbiór liter ze zmiennej letters znalezionych w zmiennej phrase."""
                                                                          ^
```

Tutaj znajduje się informacja, co jest nie tak.

Tak przy okazji: zapoznaj się z wytycznymi dotyczącymi stylu kodu Pythona, które zostały pięknie przedstawione pod adresem *http://pep8.org/*.

Problem ten można rozwiązać, skracając treść komentarza lub też dzieląc go pomiędzy dwa wiersze. Ponieważ nie chcemy utrudnić czytania tekstu dokumentującego, wybierzmy drugie rozwiązanie: wystarczy zatem *podzielić komentarz w taki sposób, aby długość jego wiersza nie przekraczała 79 znaków*. Wprowadźmy tę poprawkę i przetestujmy nasz zmieniony kod[1].

[1] Innymi typowymi problemami, z którymi moglibyśmy się tu spotkać, są błędy związane ze spacjami występującymi na końcach wierszy. Stosując przyjęty przez wydawnictwo Helion styl wcięć kodu o szerokości trzech spacji, otrzymamy też informacje o niezgodności z zalecanym dla kodu Pythona standardem wcięć o szerokości czterech spacji — *przyp. tłum.*

Potwierdzanie zgodności naszego kodu ze standardem PEP 8

W wyniku wprowadzenia niezbędnych zmian w kodzie Pythona znajdującym się w pliku *vsearch.py* ma on postać przedstawioną poniżej.

```
def search4vowels(phrase: str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Zwraca zbiór liter ze zmiennej letters znalezionych
       w zmiennej phrase."""
    return set(letters).intersection(set(phrase))
```

Wersja modułu vsearch.py jest zgodna ze standardem PEP 8.

Jeśli tę wersję kodu sprawdzimy za pomocą wtyczki **pep8** do narzędzia **pytest**, dane wyjściowe na ekranie potwierdzą, że nie mamy już żadnych problemów ze zgodnością z wytycznymi PEP 8. Poniżej możesz zobaczyć, co ujrzeliśmy na naszym komputerze (działającym pod kontrolą systemu operacyjnego *Windows*).

Kolor zielony oznacza, że jest dobrze. W tym kodzie nie występują żadne niezgodności z wytycznymi PEP 8 :)

```
C:\WINDOWS\system32\cmd.exe

c:\Users\lsuma\Desktop\Python\mymodules>py.test --pep8 vsearch.py
============================= test session starts ==============================
platform win32 -- Python 3.6.0, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
rootdir: c:\Users\lsuma\Desktop\Python\mymodules, inifile:
plugins: pep8-1.0.6
collected 1 items

vsearch.py .

========================== 1 passed in 0.06 seconds ===========================

c:\Users\lsuma\Desktop\Python\mymodules>
```

Zgodność ze standardem PEP 8 jest rzeczą dobrą

Jeśli patrzysz na to wszystko, dziwiąc się trochę, o co ten cały szum (zwłaszcza że chodziło tu właściwie tylko o kilka białych znaków), zastanów się dobrze nad powodami, dla których miałbyś chcieć zachować zgodność z wytycznymi zebranymi w dokumencie PEP 8. Stwierdza się w nim, że *czytelność ma znaczenie* oraz że kod jest *o wiele częściej czytany niż pisany*. Z faktu, że Twój kod spełnia wymagania stawiane przez standardowy styl kodowania, wynika, iż czyta się go łatwiej, ponieważ „wygląda” on tak jak wszystko to, z czym programista przeglądający Twój kod miał do tej pory do czynienia. Spójność jest dobrą rzeczą.

To już koniec naszej dygresji na temat narzędzia pytest. Do zobaczenia w rozdziale 5.!

Od tego miejsca (i w takim stopniu, w jakim jest to praktyczne) cały kod w tej książce będzie zgodny z wytycznymi zawartymi w dokumencie PEP 8. Powinieneś zapewnić, aby Twój kod również spełniał stawiane w nim wymagania.

Kod przykładów z rozdziału 4.

```
def search4vowels(phrase: str) -> set:
    """Zwraca samogłoski znalezione we frazie podanej jako argument."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Zwraca zbiór liter ze zmiennej letters znalezionych
       w zmiennej phrase."""
    return set(letters).intersection(set(phrase))
```

To jest kod modułu vsearch.py zawierającego dwie funkcje: search4vowels oraz search4letters.

To jest kod zapisany w pliku setup.py, dzięki któremu mogliśmy przekształcić nasz moduł w dystrybucję umożliwiającą instalację.

```
from setuptools import setup

setup(
   name='vsearch',
   version='1.0',
   description='Narzędzia wyszukiwania dla książki Python. Rusz głową! ',
   author='Python. Rusz głową! Wydanie II ',
   author_email='hfpy2e@gmail.com',
   url='headfirstlabs.com',
   py_modules=['vsearch'],
)
```

```
def double(arg):
   print('Przed: ', arg)
   arg = arg * 2
   print('Po:    ', arg)


def change(arg):
   print('Przed: ', arg)
   arg.append('Więcej danych')
   print('Po:    ', arg)
```

A to jest kod modułu mystery.py, który był przyczyną sporego napięcia między Tomkiem i Sarą. Na szczęście zagadkę udało się już rozwikłać i teraz znów tworzą oni zgodny zespół programistyczny :)

Coś praktycznego

Na tym etapie znasz już Pythona wystarczająco dobrze, aby stanowić zagrożenie.

Po lekturze czterech pierwszych rozdziałów tej książki jesteś już w stanie produktywnie korzystać z Pythona w wielu różnych obszarach zastosowań (nawet pomimo tego, że nadal masz sporo do nauki). Zamiast wymieniać te liczne obszary zastosowań, w tym rozdziale i kilku kolejnych mamy zamiar skonstruować nasz wykład wokół tworzenia aplikacji WWW, a więc na polu, na którym język Python sprawdza się szczególnie dobrze. Doskonaląc swoje praktyczne umiejętności programowania, dowiesz się trochę więcej o Pythonie. Zanim jednak przejdziemy dalej, zróbmy szybki przegląd tego, co wiesz już na temat tego języka.

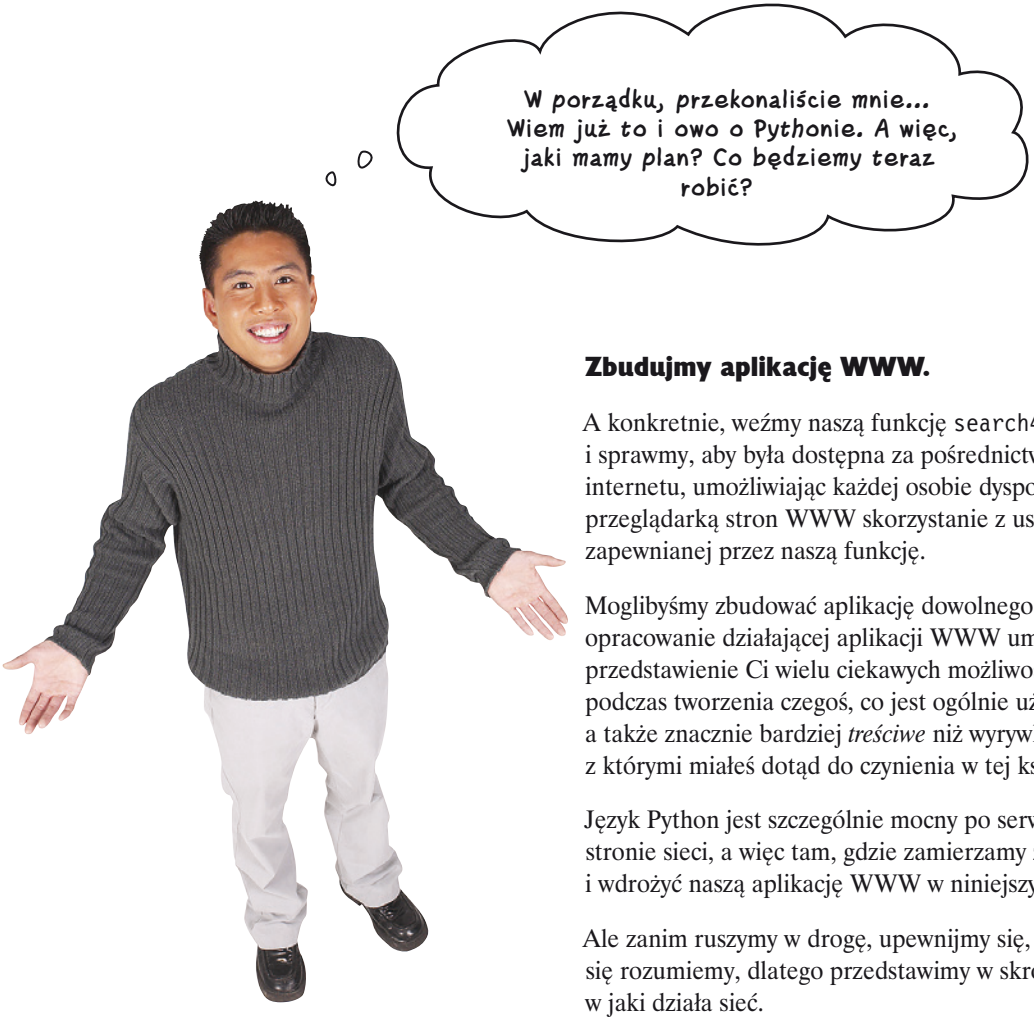
Python: co już wiemy

Teraz, gdy masz już za sobą cztery rozdziały, zatrzymajmy się na chwilę i zróbmy przegląd zaprezentowanego do tej pory materiału na temat Pythona.

CELNE SPOSTRZEŻENIA

- Wbudowanego IDE Pythona o nazwie IDLE używa się do eksperymentowania z kodem Pythona oraz uruchamiania go zarówno w postaci fragmentów składających się z pojedynczych instrukcji, jak i większych, wielowierszowych programów napisanych za pomocą edytora tekstowego zapewnianego przez środowisko. Oprócz korzystania z IDLE jest też możliwość uruchamiania kodu Pythona bezpośrednio z poziomu wiersza poleceń systemu operacyjnego za pomocą polecenia `py -3` (w systemie Windows) lub `python3` (na pozostałych platformach).
- Dowiedziałeś się, jak Python obsługuje elementy danych składające się z pojedynczych wartości, takich jak liczby całkowite czy łańcuchy znakowe, jak również wartości logiczne `True` i `False`.
- Poznałeś przypadki zastosowania czterech wbudowanych struktur danych: list, słowników, zbiorów i krotek. Wiesz już, że możesz tworzyć złożone struktury danych, łącząc ze sobą te cztery wbudowane typy w dowolny sposób.
- Używałeś sporej kolekcji instrukcji Pythona, w tym `if`, `elif`, `else`, `return`, `for`, `from` oraz `import`.
- Wiesz, że Python zapewnia bogatą bibliotekę standardową, i widziałeś w akcji następujące moduły: `datetime`, `random`, `sys`, `os`, `time`, `html`, `pprint`, `setuptools` oraz `pip`.
- Oprócz biblioteki standardowej Python oferuje wygodną kolekcję wbudowanych funkcji, znanych jako BIF. Oto kilka tego rodzaju funkcji, z których już korzystałeś: `print`, `dir`, `help`, `range`, `list`, `len`, `input`, `sorted`, `dict`, `set`, `tuple` oraz `type`.
- Python zapewnia obsługę wszystkich typowych operatorów i kilku dodatkowych. Do tych, z którymi miałeś już do czynienia, należą: `in`, `not in`, `+`, `-`, `=` (przypisanie), `==` (równość), `+=` oraz `*`.
- Oprócz zapewnienia obsługi notacji nawiasów kwadratowych (to znaczy `[]`) umożliwiającej korzystanie z elementów należących do sekwencji język Python oferuje też możliwość używania tej notacji do tworzenia **wycinków**, w przypadku których da się określić wartości **początku**, **końca** oraz **kroku**.
- Nauczyłeś się tworzyć własne funkcje w języku Python, korzystając z instrukcji `def`. Funkcje Pythona mogą przyjmować dowolną liczbę argumentów, a także zwracać wartość.
- W Pythonie łańcuchy znakowe można umieszczać w znakach pojedynczego lub podwójnego cudzysłowu, jednak zgodnie ze stosowaną w tym języku konwencją (udokumentowaną w **PEP 8**) należy wybrać jeden styl i trzymać się go konsekwentnie. W tej książce postanowiliśmy umieszczać wszystkie nasze łańcuchy znakowe w pojedynczych cudzysłowach, chyba że sam tworzony łańcuch zawiera takie znaki, wówczas używamy znaków podwójnego cudzysłowu (jednak tylko w tym jednym, szczególnym przypadku).
- Python obsługuje też łańcuchy znakowe umieszczone w potrójnych cudzysłowach. Przekonałeś się, jak można z nich korzystać, aby dodawać teksty dokumentujące do swoich funkcji.
- Poznałeś sposób grupowania powiązanych ze sobą funkcji za pomocą modułów, które stanowią podstawę mechanizmu ponownego użycia kodu w Pythonie. Dowiedziałeś się też, jak za pomocą modułu `pip` (należącego do biblioteki standardowej) można spójnie zarządzać zainstalowanymi modułami.
- Skoro już jesteśmy przy rzeczach działających w sposób spójny, warto też wspomnieć, że dowiedziałeś się, iż w Pythonie **wszystko jest obiektem**, dzięki czemu — na tyle, na ile jest to możliwe — wszystko działa tak, jak się tego spodziewasz. Rozwiązanie to naprawdę procentuje, gdy zaczyna się definiować własne obiekty za pomocą klas; pokażemy Ci, jak się to robi, w jednym z kolejnych rozdziałów tej książki.

Zbudujmy coś



Zbudujmy aplikację WWW.

A konkretnie, weźmy naszą funkcję `search4letters` i sprawmy, aby była dostępna za pośrednictwem internetu, umożliwiając każdej osobie dysponującej przeglądarką stron WWW skorzystanie z usługi zapewnianej przez naszą funkcję.

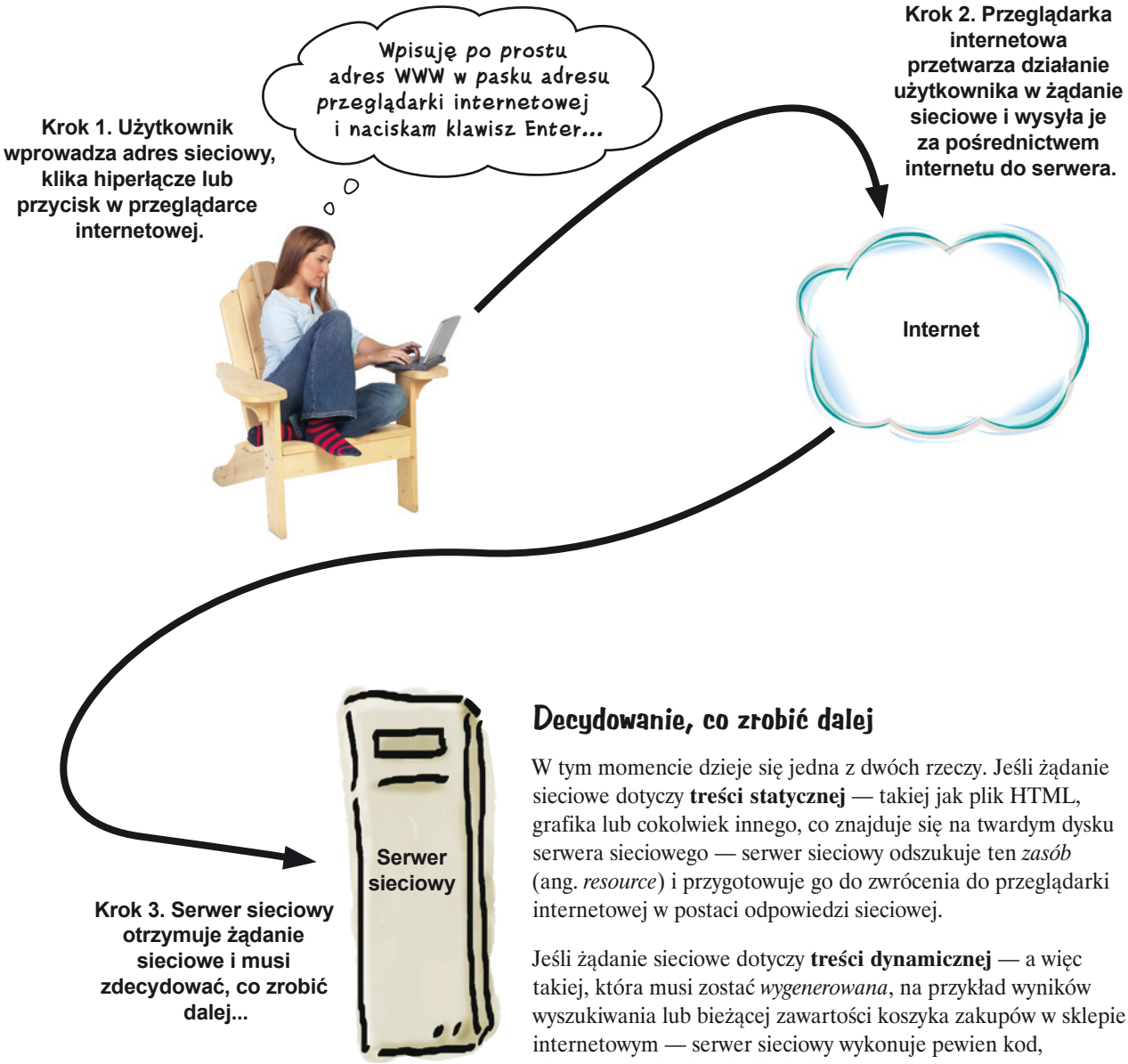
Moglibyśmy zbudować aplikację dowolnego typu, ale opracowanie działającej aplikacji WWW umożliwi nam przedstawienie Ci wielu ciekawych możliwości Pythona podczas tworzenia czegoś, co jest ogólnie użyteczne, a także znacznie bardziej *treściwe* niż wyrywki kodu, z którymi miałeś dotąd do czynienia w tej książce.

Język Python jest szczególnie mocny po serwerowej stronie sieci, a więc tam, gdzie zamierzamy zbudować i wdrożyć naszą aplikację WWW w niniejszym rozdziale.

Ale zanim ruszymy w drogę, upewnijmy się, że dobrze się rozumiemy, dlatego przedstawimy w skrócie sposób, w jaki działa sieć.

Aplikacje WWW z bliska

Niezależnie od tego, co robisz w sieci, wszystko sprowadza się tu do wymiany *żądań* (ang. *requests*) i *odpowiedzi* (ang. *responses*). **Żądanie sieciowe** (ang. *web request*) jest wysyłane z przeglądarki internetowej do serwera sieciowego w wyniku pewnego działania wykonanego przez użytkownika. Po stronie serwera sieciowego formułowana jest **odpowiedź sieciowa** (ang. *web response*), która następnie jest zwracana do przeglądarki internetowej. Cały ten proces można opisać za pomocą pięciu przedstawionych poniżej kroków.



Decydowanie, co zrobić dalej

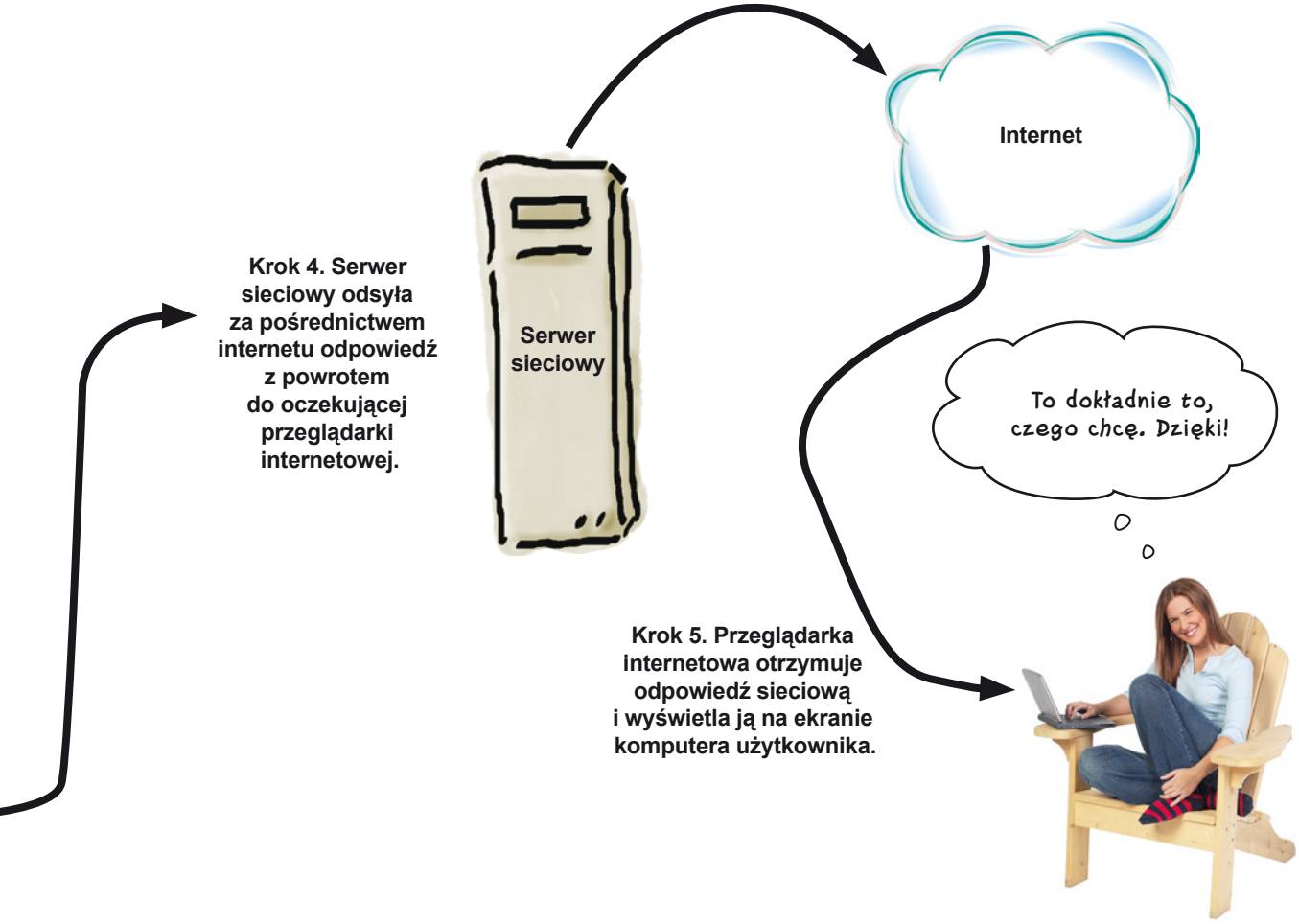
W tym momencie dzieje się jedna z dwóch rzeczy. Jeśli żądanie sieciowe dotyczy **treści statycznej** — takiej jak plik HTML, grafika lub cokolwiek innego, co znajduje się na twardym dysku serwera sieciowego — serwer sieciowy odszukuje ten *zasób* (ang. *resource*) i przygotowuje go do zwrócenia do przeglądarki internetowej w postaci odpowiedzi sieciowej.

Jeśli żądanie sieciowe dotyczy **treści dynamicznej** — a więc takiej, która musi zostać *wygenerowana*, na przykład wyników wyszukiwania lub bieżącej zawartości koszyka zakupów w sklepie internetowym — serwer sieciowy wykonuje pewien kod, aby wytworzyć odpowiedź sieciową.

(Ewentualne) liczne podkroki składowe kroku 3.

W praktyce krok 3. może wymagać wykonania wielu różnych działań w zależności od tego, co serwer sieciowy musi zrobić, aby wygenerować odpowiedź. Oczywiście, jeśli wszystko co należy zrobić, sprowadza się do znalezienia statycznej treści i zwrócenia jej do przeglądarki internetowej, niezbędne kroki składowe nie są zbyt wymagające, ponieważ wystarczy tu po prostu wykonać odpowiednie operacje odczytu danych z napędu dyskowego serwera.

Jeśli jednak zachodzi potrzeba wygenerowania treści dynamicznej, wśród kroków składowych znajdują się: uruchomienie kodu po stronie serwera sieciowego, przechwycenie odpowiednich danych wyjściowych, przetworzenie ich do postaci odpowiedzi sieciowej i odesłanie tej odpowiedzi z powrotem do oczekującej na nią przeglądarki internetowej.



Czego oczekujemy od naszej aplikacji WWW?

Choć bardzo nas kusi, aby jak zawsze *po prostu zacząć pisać kod*, powinniśmy się najpierw zastanowić, jak ma działać nasza aplikacja sieciowa.

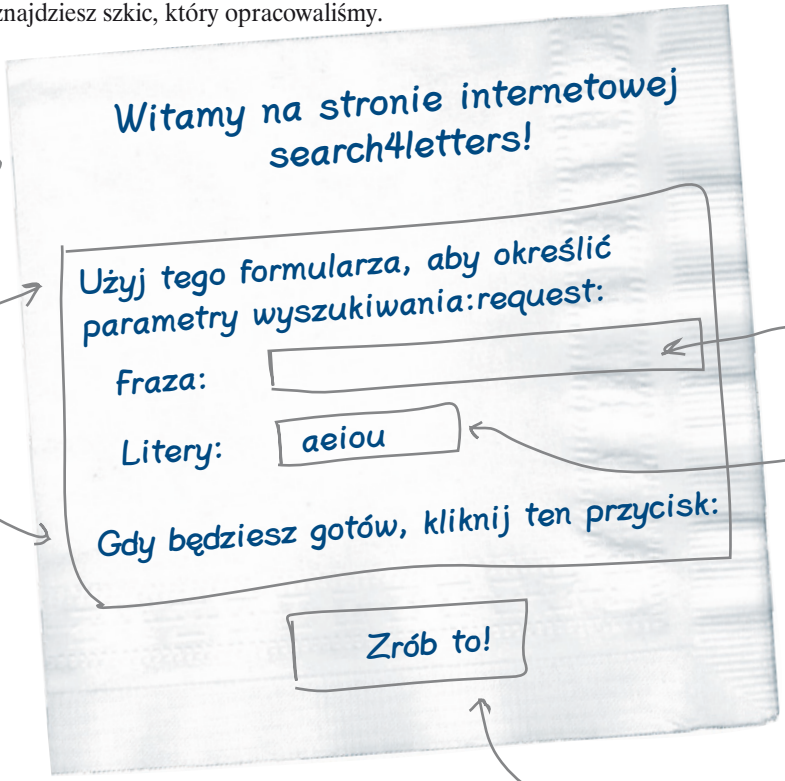
Użytkownicy będą korzystali z aplikacji WWW za pomocą swoich ulubionych przeglądarek internetowych. Aby uzyskać dostęp do świadczonych przez nią usług, muszą jedynie wprowadzić odpowiedni URL w polu adresu przeglądarki i nacisnąć klawisz *Enter*. W wyniku tego w oknie przeglądarki powinna się pojawić strona internetowa prosząca użytkownika o podanie argumentów funkcji `search4letters`. Po ich wprowadzeniu użytkownik powinien kliknąć przycisk, aby zobaczyć wyniki.

Przypomnij sobie wiersz `def` naszej najnowszej wersji funkcji `search4letters`, który określa, że spodziewa się ona co najmniej jednego — lecz nie więcej niż dwóch — argumentów: łańcucha znakowego `phrase` do przeszukania oraz łańcucha `letters` zawierającego wyszukiwane znaki. Pamiętaj, że ten ostatni jest argumentem opcjonalnym (a jego domyślna wartość to `aeiou`).

Wiersz def funkcji search4letters, która przyjmuje jeden lub dwa argumenty.

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Weźmy papierową serwetkę i naszkicujmy przewidywany wygląd naszej strony. Poniżej znajdziesz szkic, który opracowaliśmy.



Nasza strona internetowa ma tytuł i trochę opisów.

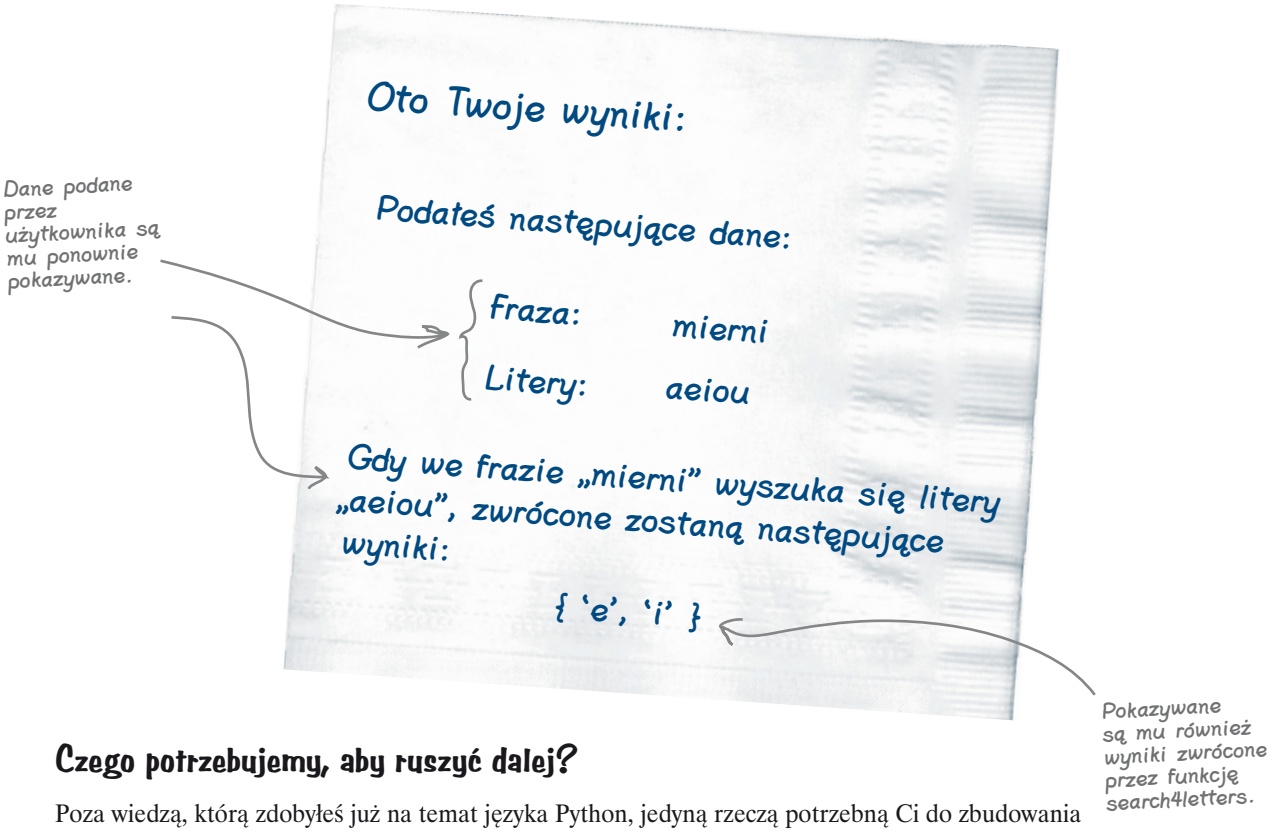
Jedno pole wprowadzania umożliwia podanie danych dla argumentu phrase, a drugie umożliwia wpisanie liter tworzących argument letters (zwróć uwagę na wartość domyślną).

Kliknięcie tego przycisku powoduje przesłanie danych wprowadzonych przez użytkownika do oczekującego na nie serwera sieciowego.

Co się dzieje po stronie serwera sieciowego?

Gdy użytkownik kliknie przycisk **Zrób to!**, przeglądarka wysyła dane do oczekującego na nie serwera sieciowego, który wydobywa wartości argumentów `phrase` i `letters`, a następnie wywołuje funkcję `search4letters` w imieniu czekającego teraz użytkownika.

Wszystkie wyniki zwrócone przez funkcję są zwracane do przeglądarki użytkownika w postaci strony internetowej, którą także naszkicowaliśmy na papierowej serwetce (pokazanej poniżej). Załóżmy na razie, że użytkownik podał słowo „mierni" dla argumentu `phrase` i pozostawił domyślną wartość `aeiou` dla argumentu `letters`. Oto jak może wyglądać nasza strona internetowa z wynikami.



Czego potrzebujemy, aby ruszyć dalej?

Poza wiedzą, którą zdobyłeś już na temat języka Python, jedyną rzeczą potrzebną Ci do zbudowania aplikacji WWW działającej po stronie serwera jest jakiś **framework aplikacji sieciowych** (ang. *web application framework*), zapewniający zbiór ogólnych, podstawowych technologii, na bazie których możesz opracować swoją aplikację.

Mimo że korzystając wyłącznie z możliwości samego Pythona, da się od zupełnych podstaw zbudować wszystko, czego tu potrzebujesz, szaleństwem byłoby już samo rozważanie takiej możliwości. Inni programiści już dawno poświęcili swój czas, aby opracować odpowiednie frameworki sieciowe, a język Python oferuje w tej kwestii duży wybór rozwiązań. Nie będziemy Cię jednak teraz zamęczać koniecznością podjęcia decyzji dotyczącej frameworka najlepszego w naszej sytuacji, lecz wybierzemy po prostu jedno z popularnych rozwiązań, noszące nazwę *Flask*, a następnie ruszymy dalej.

Zainstalujmy framework Flask

Z rozdziału 1. wiesz już, że biblioteka standardowa Pythona zapewnia mnóstwo *baterii w zestawie*. Zdarzają się jednak sytuacje, gdy musimy skorzystać ze specyficznego dla aplikacji modułu pochodzącego od osób trzecich, który *nie* stanowi części biblioteki standardowej. Tego rodzaju niezależne moduły importuje się w programach Pythona według potrzeb. Jednak w przeciwieństwie do modułów wchodzących w skład biblioteki standardowej moduły pochodzące od dostawców niezależnych trzeba zainstalować, *zanim* da się je zaimportować i wykorzystywać w swoim kodzie. Jednym z takich modułów jest Flask.

Jak wspomnieliśmy w poprzednim rozdziale, społeczność programistów Pythona utrzymuje centralnie zarządzany serwis WWW umożliwiający dostęp do modułów pochodzących od niezależnych dostawców. Serwis ten nosi nazwę **PyPI** (co jest skrótem od angielskich słów *Python Package Index*, oznaczających indeks pakietów Pythona) i przechowuje między innymi najnowszą wersję modułu Flask (jak również wiele innych projektów).

Serwis PyPI znajdziesz pod adresem *https://pypi.python.org/pypi*.

Przypomnij sobie, jak całkiem niedawno korzystaliśmy z narzędzia `pip`, aby zainstalować nasz moduł `vsearch` w Pythonie. Narzędzie to współpracuje również z serwisem PyPI. Jeśli znasz nazwę modułu, który jest Ci potrzebny, możesz skorzystać z narzędzia `pip`, aby bezpośrednio w swoim środowisku Pythona zainstalować dowolny moduł udostępniany na stronie PyPI.

Zainstaluj framework Flask z poziomu wiersza poleceń przy użyciu narzędzia pip

Jeśli korzystasz z systemu *Linux* lub *Mac OS X*, wprowadź i zatwierdź przedstawione poniżej polecenie w oknie systemowego terminala.

Użyj tego polecenia w systemach Mac OS X oraz Linux.

```
$ sudo -H python3 -m pip install flask
```

Zauważ, że wielkość liter nie ma tu znaczenia. Słowo „flask" może być napisane małą literą.

Jeśli używasz systemu *Windows*, otwórz okno wiersza poleceń — koniecznie za pomocą polecenia *Uruchom jako administrator*, które znajdziesz w menu kontekstowym dostępnym po kliknięciu programu prawym przyciskiem myszy — a następnie wprowadź i zatwierdź przedstawione poniżej polecenie.

```
C:\> py -3 -m pip install flask
```

Użyj tego polecenia w systemie Windows.

Wykonanie tego polecenia (niezależnie od używanego systemu operacyjnego) spowoduje połączenie się z serwisem PyPI, a następnie pobranie i zainstalowanie modułu **Flask** oraz czterech innych modułów, od których jest on uzależniony, czyli: **Werkzeug**, **MarkupSafe**, **Jinja2** oraz **itsdangerous**. Nie martw się (na razie) tym, co te dodatkowe moduły robią; sprawdź tylko, czy zainstalowały się prawidłowo. Jeśli wszystko poszło dobrze, na końcu danych wyjściowych wygenerowanych przez narzędzie `pip` ujrzysz komunikat podobny do przedstawionego poniżej. Zauważ, że te dane wyjściowe zajmują na ekranie nawet kilkanaście wierszy.

Widoczne tu numery określają najnowsze wersje modułów w czasie tłumaczenia tej książki.

```
...
Successfully installed Jinja2-2.9.6 MarkupSafe-1.0 Werkzeug-0.12.2 flask-0.12.2
itsdangerous-0.24
```

Jeśli na ekranie nie widzisz komunikatu zaczynającego się od słów `Successfully installed...`, sprawdź, czy Twój komputer ma połączenie z internetem i czy wprowadziłeś odpowiednie dla swojego systemu operacyjnego polecenia *dokładnie* w takiej formie, w jakiej zostało to przed chwilą przedstawione. Nie martw się, jeśli numery wersji modułów zainstalowanych w Twoim środowisku Pythona różnią się od tych, które widać powyżej (ponieważ moduły są ciągle aktualizowane, a zależności również mogą się zmieniać). Dopóki instalowane przez Ciebie wersje są *co najmniej* tak aktualne, jak te przedstawione, wszystko jest w jak najlepszym porządku.

Jak działa framework Flask?

Flask to w rzeczywistości kolekcja modułów, które pomagają budować aplikacje WWW działające po stronie serwera. Z technicznego punktu widzenia jest to *mikro*framework sieciowy w tym sensie, że zapewnia minimalny zestaw technologii potrzebnych, aby spełnić swoje zadanie. Oznacza to, że Flask nie jest tak doskonale wyposażonym frameworkiem, jak niektóre z konkurencyjnych rozwiązań — takie jak **Django**, ojciec wszystkich frameworków sieciowych opracowanych w języku Python — ale jest za to niewielki, lekki i łatwy w użyciu.

Z uwagi na to, że nasze wymagania nie są wielkie (planujemy mieć tylko dwie strony), framework Flask spełni nasze oczekiwania z nawiązką.

Porady maniaka

Django to framework aplikacji sieciowych niezwykle popularny wśród programistów Pythona. Oferuje szczególnie duże możliwości administracyjne, dzięki którym da się wydajnie tworzyć rozbudowane aplikacje WWW. Zastosowanie go w naszym przypadku byłoby grubą przesadą, dlatego postanowiliśmy wybrać znacznie prostszy, ale też o wiele lżejszy, framework **Flask**.

Sprawdź, czy framework został zainstalowany i działa

Poniżej znajdziesz kod wykorzystywany w najprostszych aplikacjach WWW tworzonych za pomocą Flaska. Skorzystamy z niego, aby sprawdzić, czy framework został poprawnie zainstalowany i jest gotowy do użycia.

Skorzystaj ze swojego ulubionego edytora tekstowego, aby utworzyć nowy plik, a następnie przepisz kod pokazany poniżej i zapisz go pod nazwą *hello_flask.py* (jeśli chcesz, plik ten możesz też zapisać w osobnym folderze; my nadaliśmy naszemu folderowi nazwę *webapp*).

Kod gotowy do użycia

To jest plik hello_flask.py.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

app.run()
```

Przepisz ten kod dokładnie w takiej formie, w jakiej został tu zaprezentowany. Wkrótce omówimy jego znaczenie.

Uruchom aplikację wykorzystującą framework Flask z poziomu wiersza poleceń swojego systemu operacyjnego

Nie ulegaj pokusie uruchamiania tego kodu z poziomu środowiska IDLE, ponieważ nie jest ono przeznaczone do takich działań i nie poradzi sobie z tym w odpowiedni sposób. IDLE sprawdza się świetnie podczas eksperymentów z kodem, ale gdy trzeba uruchamiać aplikacje, znacznie rozsądniej jest uruchamiać kod za pomocą interpretera z poziomu wiersza poleceń systemu operacyjnego. Zróbmy to teraz i zobaczmy, co się stanie.

Nie używaj środowiska IDLE do uruchamiania tego kodu.

Pierwsze uruchomienie aplikacji WWW wykorzystującej framework Flask

Jeśli korzystasz z systemu operacyjnego *Windows*, otwórz okno wiersza poleceń w folderze zawierającym Twój plik programu *hello_flask.py*. (Wskazówka: jeśli otworzyłeś swój folder w oknie programu *Eksplorator plików*, przytrzymaj klawisz *Shift* i kliknij okno prawym przyciskiem myszy, aby wywołać menu kontekstowe, a następnie wybierz z niego polecenie *Otwórz okno polecenia tutaj*). W oknie wiersza poleceń wprowadź i zatwierdź przedstawione poniżej polecenie, aby uruchomić swoją aplikację wykorzystującą framework Flask.

Nasz kod zapisaliśmy w pliku umieszczonym w folderze o nazwie „webapp".

```
C:\webapp> py -3 hello_flask.py
```

Za pomocą tego polecenia prosimy interpreter o uruchomienie kodu znajdującego się w pliku hello_flask.py.

Jeśli używasz systemu operacyjnego *Mac OS X* lub *Linux*, wpisz przedstawione tu polecenie w oknie terminala. Pamiętaj, aby zrobić to w folderze zawierającym Twój plik programu *hello_flask.py*.

```
$ python3 hello_flask.py
```

Niezależnie od tego, z jakiego systemu operacyjnego korzystasz, framework Flask przejmuje od tego momentu kontrolę nad wyświetlaniem komunikatów stanu na ekranie za każdym razem, gdy wbudowany serwer WWW wykonuje jakąś operację. Bezpośrednio po rozpoczęciu pracy serwer Flaska potwierdza, że wystartował i działa oraz że oczekuje na żądania sieciowe pod adresem testowym (`127.0.0.1`) i numerem portu protokołu (5000).

Jeśli widzisz ten komunikat, wszystko jest w porządku.

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Serwer WWW frameworka Flask jest gotowy i oczekuje. *Co teraz?* Spróbujmy wejść w interakcję z serwerem, korzystając z naszej przeglądarki internetowej. Otwórz dowolny program tego rodzaju i wprowadź w jego polu adresu URL widoczny w pierwszym komunikacie serwera Flaska.

To adres, pod którym dostępna jest Twoja aplikacja WWW. Wprowadź go dokładnie w takiej formie, w jakiej jest tu widoczny.

```
http://127.0.0.1:5000/
```

Po chwili w oknie przeglądarki internetowej powinien pojawić się komunikat „Witaj, świecie, tu Flask!", pochodzący z pliku *hello_flask.py*. To jednak jeszcze nie wszystko. Spójrz teraz w okno terminala, w którym działa Twoja aplikacja. Powinien się w nim pojawić przedstawiony poniżej komunikat stanu.

Aha! Coś się tu stało.

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Jun/2017 16:43:17] "GET / HTTP/1.1" 200
```

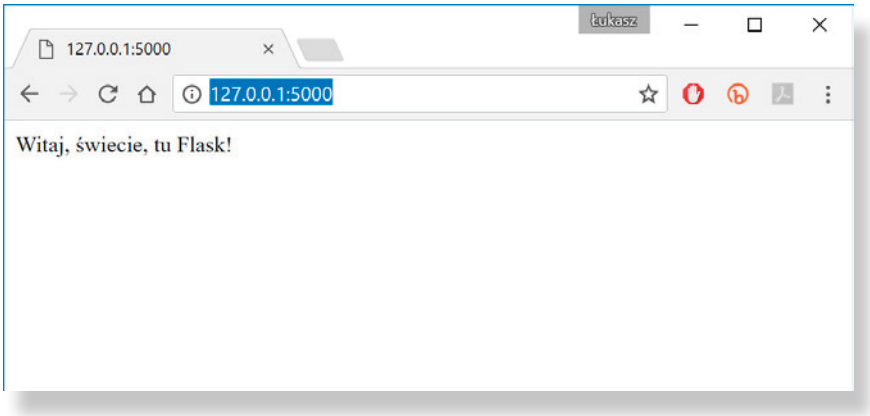
Porady maniaka

Wgłębianie się w szczegóły dotyczące znaczenia **numeru portu protokołu** wykracza poza zakres tematyczny tej książki. Jeśli jednak chciałbyś dowiedzieć się czegoś więcej na ten temat, zacznij lekturę od tego miejsca: ***https://pl.wikipedia.org/wiki/Port_protokołu***.

Oto co się stało (wiersz po wierszu)

Oprócz tego, że w oknie terminala pojawiają się wyświetlane przez framework Flask informacje o aktualnym stanie, przeglądarka internetowa pokazuje też odpowiedź uzyskaną od serwera WWW. Poniżej możesz zobaczyć, jak wygląda ona w naszej przeglądarce (tutaj jest to program *Chrome* działający pod kontrolą systemu operacyjnego *Windows*).

Tutaj widoczny jest komunikat zwrócony przez serwer WWW frameworka Flask.



Gdy użyliśmy naszej przeglądarki internetowej, aby przejść pod adres wskazywany przez URL widoczny w początkowym komunikacie stanu, serwer odpowiedział komunikatem o treści: „Witaj, świecie, tu Flask!”.

Mimo że nasza aplikacja WWW składa się z zaledwie sześciu wierszy kodu, dzieje się tu całkiem sporo, dlatego przeanalizujmy kod wiersz po wierszu, aby przekonać się, jakie dokładnie działania są wykonywane przez aplikację. Wszystko, co mamy zamiar zbudować, powstanie na bazie tych sześciu wierszy kodu.

W pierwszym wierszu z modułu importowana jest klasa `Flask` należąca do modułu `flask`.

To jest nazwa modułu: flask (pisana małą literą).

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

app.run()
```

To jest nazwa klasy: Flask (pisana wielką literą).

Pamiętasz, jak omawialiśmy różne sposoby importowania?

Mógłbyś również skorzystać tu z instrukcji `import flask`, a następnie odwoływać się do klasy `Flask`, korzystając z zapisu `flask.Flask`, ale użycie wersji instrukcji `import` ze słowem kluczowym `from` jest w tym przypadku preferowanym rozwiązaniem, ponieważ zapis `flask.Flask` znacznie trudniej się czyta.

Tworzenie obiektu aplikacji WWW frameworka Flask

W drugim wierszu kodu tworzony jest obiekt typu `Flask`, który zostaje następnie przypisany do zmiennej `app`. Wydaje się to dość proste; wyjątkiem jest tu zastosowanie nieco dziwnego argumentu o nazwie `__name__`.

Utwórz instancję klasy Flask i przypisz ją do zmiennej app.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

app.run()
```

O co tu chodzi?

Wartość `__name__` jest utrzymywana przez interpreter Pythona i w momencie użycia w kodzie programu zawsze zawiera nazwę aktualnie aktywnego modułu. Okazuje się, że klasa `Flask` musi znać bieżącą wartość `__name__` w chwili tworzenia nowego obiektu, dlatego nazwa ta powinna zostać przekazana jako argument. I to właśnie z tego powodu posłużyliśmy się nią w tym miejscu (mimo że w istocie wygląda to dość *dziwnie*).

Ten pojedynczy wiersz kodu, mimo że dość krótki, wykonuje mnóstwo dobrej roboty, ponieważ framework Flask ukrywa przed Twoimi oczami wiele szczegółów dotyczących programowania sieciowego. Umożliwia Ci to skoncentrowanie się na definiowaniu działania będącego odpowiedzią na pojawienie się żądania sieciowego na serwerze. Właśnie tym zajmiemy się już w następnym omawianym wierszu kodu.

Porady maniaka

Zauważ, że nazwa `__name__` składa się z dwóch znaków podkreślenia, po których występuje słowo „name", po którym z kolei występują kolejne dwa znaki podkreślenia. Określa się je mianem „podwójnych podkreśleń" (ang. *double underscores*), gdy są wykorzystywane w roli przedrostka i przyrostka nazwy w kodzie Pythona. Z tą konwencją nazw będziesz miał nieraz do czynienia w swoich podróżach z Pythonem. Często też będziesz się spotykać z określeniem „dunder name", będącym **skrótem** opisowego wyrażenia „podwójne podkreślenie, name, podwójne podkreślenie". Takiego właśnie skrótowego określenia wolą używać w takich sytuacjach oszczędni programiści Pythona. Jako że podwójne podkreślenia stosowane są w Pythonie bardzo powszechnie, znane są też jako „dundery". W dalszej części tej książki zobaczysz jeszcze mnóstwo innych dunderów oraz przykładów ich użycia.

Oprócz dunderów korzysta się też z innej konwencji, polegającej na poprzedzaniu właściwej nazwy zmiennej pojedynczym znakiem podkreślenia. Niektórzy programiści Pythona określają tak utworzone nazwy analogicznym mianem „wonder", stanowiącym skrót od angielskich słów „one underscore", oznaczających jeden znak podkreślenia.

Dekorowanie funkcji za pomocą ciągu URL

W kolejnym wierszu pojawia się nowy kawałek składni Pythona: **dekoratory**. Dekorator funkcji, z którym mamy do czynienia w tym kodzie, dostosowuje działanie istniejącej funkcji *bez* konieczności zmiany jej kodu (tj. funkcji poddanej dekoracji).

Niewykluczone, że to ostatnie zdanie zechcesz przeczytać kilka razy.

W istocie dekoratory umożliwiają Ci wzięcie pewnego istniejącego kodu i dołożenie do niego jakiegoś dodatkowego zachowania zgodnie z potrzebą. Dekoratory mogą być używane zarówno w stosunku do klas, jak i funkcji, lecz stosuje się je przede wszystkim do tych drugich, dlatego większość programistów Pythona określa je mianem **dekoratorów funkcji**.

Przyjrzyjmy się dekoratorowi funkcji występującemu w kodzie naszej aplikacji WWW, który bardzo łatwo jest znaleźć, ponieważ zaczyna się symbolem @, co pokazano poniżej.

Porady maniaka

Stosowana w Pythonie składnia dekoratorów została zainspirowana używaną w języku Java składnią adnotacji, jak również konstrukcjami wykorzystywanymi w świecie programowania funkcyjnego.

Tutaj znajduje się dekorator funkcji, który — jak każdy dekorator — zaczyna się symbolem @.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

app.run()
```

To jest URL.

Mimo że da się tworzyć własne dekoratory funkcji (o czym będzie mowa w jednym z kolejnych rozdziałów tej książki), teraz skoncentrujemy się wyłącznie na używaniu gotowych. W języku Python istnieje mnóstwo wbudowanych dekoratorów, a wiele modułów pochodzących z niezależnych źródeł (takich jak właśnie Flask) zapewnia dekoratory specjalnego zastosowania (jednym z nich jest `route`).

Kod Twojej aplikacji WWW może uzyskać dostęp do zapewnianego przez framework Flask dekoratora `route` za pośrednictwem zmiennej `app`, która została utworzona w poprzednim wierszu programu.

Dekorator `route` umożliwia powiązanie ścieżki sieciowej URL z istniejącą funkcją Pythona. W tym przypadku URL o postaci „/” został powiązany z funkcją o nazwie `hello` zdefiniowaną w kolejnym wierszu kodu. Dekorator `route` sprawia, że serwer WWW Flask wywołuje tę funkcję, gdy serwer ten otrzyma żądanie URL-a „/”. Dekorator czeka wówczas na wynik wytworzony przez dekorowaną funkcję, a następnie zwraca dane wyjściowe do serwera, który z kolei przekazuje je do oczekującej przeglądarki internetowej.

Nie musimy znać sposobu, w jaki Flask (oraz dekorator `route`) czyni całą opisaną powyżej „magię”. Ważne jest, że framework robi to dla nas, a my musimy jedynie napisać funkcję, która zwraca potrzebny nam wynik. O resztę dbają już Flask i dekorator `route`.

Dekorator funkcji dostosowuje zachowanie istniejącej funkcji (bez modyfikowania jej kodu).

Uruchamianie działań Twojej aplikacji WWW

Gdy mamy już napisany wiersz dekoratora `route`, w następnym wierszu należy rozpocząć dekorowaną funkcję. W naszej aplikacji WWW jest to funkcja `hello`, która wykonuje tylko jedną operację: gdy zostanie wywołana, zwraca komunikat „Witaj, świecie, tu Flask!”.

To tylko zwykła funkcja Pythona, która — gdy się ją wywoła — zwraca łańcuch tekstowy do miejsca wywołania (zwróć uwagę na adnotację -> str).

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

app.run()
```

W ostatnim wierszu kodu obiekt Flaska przypisany do zmiennej `app` jest proszony o uruchomienie serwera sieciowego. Robi się to, wywołując metodę `run`.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

app.run()
```

Tutaj prosimy aplikację o rozpoczęcie pracy.

Na tym etapie Flask uruchamia swój wbudowany serwer WWW i wykonuje na nim kod Twojej aplikacji. Na wszelkie otrzymane przez serwer żądania dotyczące URL-a „/” udzielana jest odpowiedź mająca postać komunikatu „Witaj, świecie, tu Flask!”, a pojawienie się żądania jakiegokolwiek innego URL-a skutkuje komunikatem błędu „Resource not found” informującym, że żądany zasób nie został znaleziony. Aby zobaczyć mechanizm obsługi błędów w akcji, wprowadź przedstawiony poniżej ciąg URL w polu adresu swojej przeglądarki internetowej i naciśnij klawisz *Enter*.

`http://127.0.0.1:5000/czytodziala.html`

W oknie Twojej przeglądarki pojawi się komunikat „Not Found” (nie znaleziono), a Twoja aplikacja WWW działająca w oknie terminala zaktualizuje stan, wyświetlając odpowiedni komunikat.

Ten URL nie istnieje: błąd 404!

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Jun/2017 16:43:17] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Jun/2017 20:12:05] "GET /czytodziala.html HTTP/1.1" 404 -
```

Komunikat, który tu widzisz, może być nieco inny. Nie powinno Cię to jednak martwić.

Udostępnianie funkcji w sieci

Biorąc pod uwagę fakt, że właśnie zbudowałeś działającą aplikację WWW, której kod ogranicza się do zaledwie sześciu wierszy, zastanów się, co zapewnia Ci tu framework Flask. Oferuje mianowicie mechanizm, dzięki któremu możesz wziąć dowolną istniejącą funkcję Pythona i wyświetlić zwracany przez nią wynik w przeglądarce internetowej.

Aby zwiększyć możliwości swojej aplikacji WWW, powinieneś wybrać ciąg URL, który chcesz powiązać z określoną funkcją, a następnie napisać odpowiedni dekorator `@app.route` w wierszu znajdującym się powyżej funkcji wykonującej rzeczywistą pracę. Zróbmy to teraz, korzystając z możliwości zapewnianej przez funkcję `search4lettters`, którą napisaliśmy w poprzednim rozdziale.

Zaostrz ołówek

Zmodyfikujmy kod programu *hello_flask.py* w taki sposób, aby obsługiwał drugi ciąg URL: `/search4`. Napisz kod wiążący ten URL z funkcją o nazwie `do_search`, która wywołuje z kolei funkcję `search4letters` (należącą do naszego modułu `vsearch`). Następnie spraw, aby funkcja `do_search` zwracała rezultaty uzyskane w wyniku wyszukiwania we frazie „życie, wszechświat i cała reszta" znaków wchodzących w skład łańcucha `'eiru,!'`.

Poniżej przedstawiony został nasz istniejący kod, w którym zarezerwowaliśmy trochę miejsca dla nowych instrukcji, jakie powinieneś napisać. Twoim zadaniem jest opracowanie brakującego kodu.

Wskazówka: wyniki zwracane przez funkcję `search4letters` są zbiorem Pythona. Pamiętaj, aby przeprowadzić rzutowanie tych wyników na łańcuch znakowy, wywołując BIF `str`, *zanim* zwrócisz cokolwiek oczekującej przeglądarce internetowej, ponieważ spodziewa się ona otrzymać dane tekstowe, a nie zbiór Pythona. (Gdybyś zapomniał: skrót „BIF" oznacza w Pythonie *funkcję wbudowaną*).

Czy musisz coś zaimportować?

```
from flask import Flask
.....................................................................
.....................................................................

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'
.....................................................................
.....................................................................
.....................................................................
.....................................................................

app.run()
```

Dodaj drugi dekorator.

Dodaj kod funkcji do_search w tym miejscu.

Zaostrz ołówek
Rozwiązanie

Miałeś zmodyfikować kod programu *hello_flask.py* w taki sposób, aby obsługiwał drugi ciąg URL, `/search4`. Twoje zadanie polegało na napisaniu kodu wiążącego ten URL z funkcją o nazwie `do_search`, która sama wywołuje z kolei funkcję `search4letters` (należącą do naszego modułu `vsearch`). Następnie miałeś sprawić, aby funkcja `do_search` zwracała rezultaty uzyskane w wyniku wyszukiwania we frazie „życie, wszechświat i cała reszta" znaków wchodzących w skład łańcucha `'eiru,!'`.

Poniżej przedstawiony został nasz istniejący kod, w którym zarezerwowaliśmy trochę miejsca dla nowych instrukcji, jakie powinieneś napisać. Twoim zadaniem było opracowanie brakującego kodu.

Jak wyniki Twojej pracy wyglądają w porównaniu z naszymi?

Musisz zaimportować funkcję search4letters z modułu vsearch, zanim będziesz mógł ją wywołać.

```
from flask import Flask

from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('życie, wszechświat i cała reszta', 'eiru,!'))

app.run()
```

Drugi dekorator ustawia URL /search4.

Funkcja do_search wywołuje funkcję search4letters, a następnie zwraca jej wynik w postaci łańcucha znakowego.

Aby przetestować tę nową funkcję, musisz ponownie uruchomić swoją aplikację WWW wykorzystującą framework Flask, ponieważ obecnie wykonuje ona poprzednią wersję kodu. Aby ją zatrzymać, powróć do okna terminala, a następnie użyj kombinacji klawiszy *Ctrl+C*. Twoja aplikacja WWW zakończy działanie i wrócisz do wiersza poleceń systemu operacyjnego. Naciśnij klawisz strzałki skierowanej w górę, aby przywołać poprzednie polecenie (to, z którego korzystałeś wcześniej, aby uruchomić program *hello_flask.py*), a następnie naciśnij klawisz *Enter*. Początkowy komunikat stanu Flaska pojawi się ponownie, aby potwierdzić, że Twoja zaktualizowana aplikacja oczekuje na żądania, tak jak zostało to przedstawione poniżej.

```
C:\Users\lsuma\Desktop\Python\R05>py -3 hello_flask.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Jun/2017 16:43:17] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Jun/2017 20:12:05] "GET /czytodziala.html HTTP/1.1" 404 -
^C
C:\Users\lsuma\Desktop\Python\R05>py -3 hello_flask.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Zatrzymaj aplikację WWW...

...następnie ponownie ją uruchom.

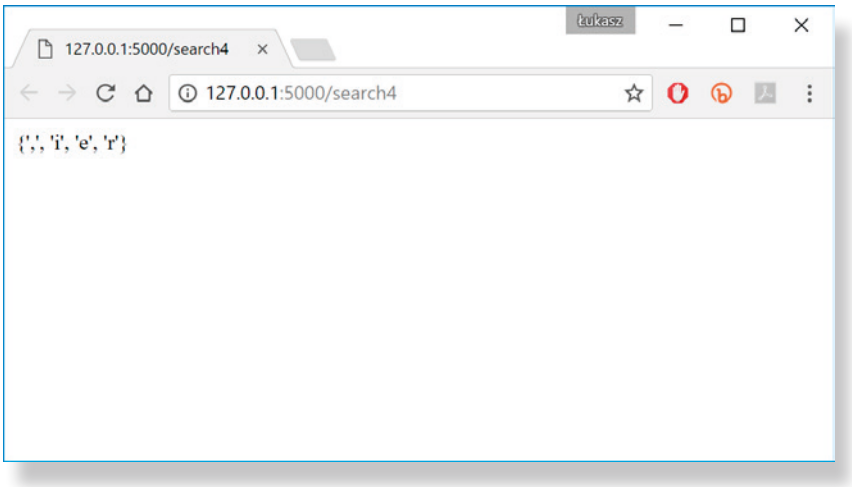
Aplikacja została ponownie uruchomiona i działa.

Jazda próbna

Jako że nie zmieniałeś kodu powiązanego ze standardowym URL-em „/”, działa on nadal, wyświetlając niezmiennie komunikat „Witaj, świecie, tu Flask!”.

Gdy jednak wprowadzisz w pasku adresu przeglądarki internetowej ciąg ***http://127.0.0.1:5000/search4*** i zatwierdzisz go za pomocą klawisza *Enter*, zobaczysz w tym oknie wyniki wywołania metody `search4letters`.

Tu widać wyniki wywołania funkcji search4letters. W komunikacie tym nie ma oczywiście niczego szczególnie ekscytującego, ale potwierdza on, że użycie ciągu URL /search4 powoduje wywołanie naszej funkcji i zwrócenie przez nią wyników.



Nie istnieją głupie pytania

P: Nie bardzo rozumiem znaczenia części składowych ciągu URL, tj. `127.0.0.1` oraz `:5000`. O co tu chodzi?

O: W tej chwili testujesz Twoją aplikację WWW na swoim komputerze, który ma własny, unikatowy adres IP, ponieważ jest podłączony do internetu. Mimo to Flask nie korzysta z tego adresu IP, lecz łączy się z testowym serwerem WWW, używając **adresu pętli zwrotnej** (ang. *loopback address*) `127.0.0.1`, znanego również jako `localhost`. Obydwa adresy stanowią wygodne skróty oznaczające: „mój komputer, niezależnie od jego rzeczywistego adresu IP”. Aby Twoja przeglądarka internetowa (również działająca na Twoim komputerze) mogła komunikować się z Twoim serwerem sieciowym Flaska, musisz podać adres, pod którym dostępna jest Twoja aplikacja WWW, czyli właśnie `127.0.0.1`. Jest to standardowy adres IP zarezerwowany konkretnie w tym celu.

Część `:5000` ciągu URL określa **numer portu protokołu**, z którego korzysta Twój serwer WWW.

Serwery WWW używają zwykle portu protokołu 80, który stanowi internetowy standard i jako taki nie musi być określany. Możesz więc wprowadzić w polu adresu przeglądarki internetowej ciąg `helion.pl:80` i zobaczysz stronę, o którą Ci chodzi. Nikt jednak tak nie robi, ponieważ wpisanie samego `helion.pl` jest zupełnie wystarczające (jako że `:80` jest dodawane domyślnie).

Gdy buduje się aplikację WWW, bardzo rzadko wykorzystuje się port 80 do testów (ponieważ jest on zarezerwowany dla serwerów produkcyjnych), dlatego większość frameworków sieciowych używa w tym celu innych portów. Popularnym portem jest tu 8080, jednak w przypadku Flaska wykorzystywany jest port 5000.

P: Czy mogę korzystać z innego portu niż 5000, gdy testuję i uruchamiam swoją aplikację wykorzystującą framework Flask?

O: Tak, metoda `app.run()` umożliwia Ci określenie wartości `port`, która może być ustawiona dowolnie. Jeśli jednak nie masz naprawdę solidnego powodu, aby dokonać tej zmiany, trzymaj się na razie standardowej wartości 5000.

Przypomnij sobie, co próbujemy zbudować

Nasza aplikacja WWW potrzebuje strony internetowej, która będzie przyjmować dane wejściowe, a także drugiej, która będzie wyświetlać wyniki wygenerowane na podstawie tych danych przez funkcję `search4letters`. Nasza aktualna wersja kodu nawet nie zbliża się do tego celu, ale to, co mamy, stanowi podstawę, na której można budować to, co *jest* wymagane.

Poniżej, po lewej stronie, przedstawiliśmy nasz bieżący kod, natomiast po prawej — kopie „serwetkowych specyfikacji", które opracowaliśmy we wcześniejszej części tego rozdziału. Jak widać, wskazaliśmy miejsca w kodzie, w których — jak sądzimy — mogą być zapewnione możliwości związane z treścią każdej z serwetek.

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters( ... ))

app.run()
```

Aby wszystko zmieścić, nie pokazujemy tu całego wiersza kodu.

Witamy na stronie internetowej search4letters!

Użyj tego formularza, aby określić parametry wyszukiwania:

Fraza:

Litery: aeiou

Gdy będziesz gotów, kliknij ten przycisk:

Zrób to!

Oto Twoje wyniki:

Podałeś następujące dane:

Fraza: mierni

Litery: aeiou

Gdy we frazie „mierni" wyszuka się litery „aeiou", zwrócone zostaną następujące wyniki:

{ 'e', 'i' }

Oto nasz plan

Zmienimy funkcję `hello` w taki sposób, aby zwracać wyniki w formie kodu HTML. Następnie zmodyfikujemy funkcję `do_search` tak, aby przyjmowała dane wejściowe z formularza przed wywołaniem funkcji `search4letters`. Wyniki będą następnie zwracane przez funkcję `do_search` w postaci kolejnej strony internetowej.

Budowanie formularza HTML

Wymagany formularz HTML nie jest szczególnie skomplikowany. Poza kilkoma tekstami opisowymi powinien zawierać jedynie dwa pola wprowadzania oraz przycisk.

Ale... co jeśli ten cały HTML jest dla Ciebie czymś nowym?

Nie panikuj, jeśli cała ta gadka o formularzu HTML, polach wprowadzania i przyciskach przyprawia Cię o dreszcze. Nie bój się, bo mamy dla Ciebie coś, czego szukasz: książka *HTML5. Rusz głową!* dostarczy Ci podstawowych informacji na temat tej technologii, jeśli potrzebujesz do niej wprowadzenia (lub chcesz szybko odświeżyć swoją wiedzę).

Jeśli myśl o odłożeniu niniejszej książki na bok w celu rozpoczęcia lektury innej, poświęconej językowi HTML, nie wydaje Ci się rozsądna i obawiasz się, że to zbyt dużo zachodu, wiedz, że podamy Ci tu cały kod HTML niezbędny do pracy z przykładami zamieszczonymi w naszym podręczniku i wcale nie musisz mieć eksperckiej wiedzy na temat tego języka. Pewne doświadczenie w pracy z HTML-em pomoże, ale nie stanowi bezwzględnego wymogu (w końcu jest to książka o Pythonie, nie o HTML-u).

Uwaga z działu marketingu: to książka, którą całym sercem polecamy osobom pragnącym szybko rozpocząć swoją przygodę z językiem HTML5. Nie żebyśmy byli stronniczy czy coś :)

Utwórz kod HTML, następnie prześlij go do przeglądarki

Zawsze istnieje wiele możliwości wykonania tego samego zadania, a gdy chodzi o tworzenie tekstu HTML z poziomu aplikacji WWW korzystającej z frameworka Flask, do wyboru są dwie przedstawione poniżej opcje.

Lubię umieszczać swoje HTML-e w wielkich łańcuchach znakowych, które osadzam następnie w swoim kodzie Pythona i zwracam odpowiednie łańcuchy, jeśli jest to konieczne. Dzięki temu wszystko, czego potrzebuję, mam pod ręką w swoim kodzie i sprawuję nad tym pełną kontrolę... A właśnie to jest mój ulubiony sposób działania. Co może się w tym nie podobać, Lauro?

Cóż, Robercie, umieszczenie całego HTML-a w Twoim kodzie z pewnością się sprawdzi, ale nie będzie się to dobrze skalować. Gdy Twoja aplikacja WWW zacznie się rozrastać, cały ten osadzony kod HTML narobi niezłego bałaganu... Poza tym trudno go będzie przekazać projektantowi stron WWW, aby mógł go on upiększyć. Nie będzie też łatwo ponownie używać jego fragmentów. Dlatego w swoich aplikacjach WWW zawsze korzystam z silnika szablonów (ang. *template engine*). Wiąże się to z koniecznością wykonania dodatkowej pracy na początku, ale z czasem inwestycja ta naprawdę zaczyna się opłacać...

Laura ma rację. Szablony sprawią, że kod HTML stanie się znacznie łatwiejszy w utrzymaniu, niż ma to miejsce w przypadku rozwiązania preferowanego przez Roberta. Szablonami zajmiemy się już na następnej stronie.

Szablony z bliska

Silniki szablonów (zwane też systemami szablonów) umożliwiają programistom stosowanie zorientowanych obiektowo koncepcji dziedziczenia i ponownego użycia przy tworzeniu danych tekstowych, takich jak strony internetowe.

Wygląd i sposób działania strony WWW może być definiowany za pomocą szablonu HTML najwyższego poziomu znanego jako **szablon bazowy** (ang. *base template*), który jest następnie dziedziczony przez inne strony HTML. Jeśli wprowadzisz zmianę w szablonie bazowym, zostanie ona odzwierciedlona na *wszystkich* stronach HTML, które po nim dziedziczą.

Silnik szablonów dostarczany wraz z frameworkiem Flask nosi nazwę *Jinja2*; rozwiązanie to jest równocześnie proste w użyciu i bardzo potężne. W niniejszej książce nie mamy zamiaru uczyć Cię wszystkiego, co powinieneś wiedzieć o systemie Jinja2, dlatego treść, która pojawi się na tych dwóch stronach, jest z konieczności krótka i rzeczowa. Więcej szczegółów na temat tego, co można osiągnąć za pomocą silnika Jinja2, znajdziesz pod podanym adresem:

http://jinja.pocoo.org/docs/dev/

Poniżej zaprezentowaliśmy bazowy styl, z którego będziemy korzystać w naszej aplikacji WWW. W tym pliku, noszącym nazwę *base.html*, umieściliśmy znaczniki HTML, które mają być używane w przypadku naszych wszystkich stron internetowych. Zastosowaliśmy również kilka znaczników specyficznych dla silnika Jinja2, aby wskazać treść, która zostanie dostarczona, gdy dziedziczące po tym szablonie strony HTML będą renderowane (tzn. przygotowywane przed wysłaniem do oczekującej przeglądarki internetowej). Zwróć uwagę na to, że znaczniki ujęte w znaki {{ oraz }} oraz te umieszczone pomiędzy znakami [% i %] są przeznaczone dla silnika szablonów Jinja2. Poniżej zaznaczyliśmy wystąpienia tych znaczników, aby łatwiej Ci było je zauważyć.

```
<!doctype html>
<html>
    <head>
        <title>{{ the_title }}</title>
        <link rel="stylesheet" href="static/hf.css" />
    </head>
    <body>
        {% block body %}

        {% endblock %}
    </body>
</html>
```

To standardowy znacznik języka HTML5.

To jest dyrektywa silnika Jinja2, która wskazuje, że ta wartość zostanie dostarczona przed wyrenderowaniem strony (możesz o niej myśleć jako o argumencie szablonu).

To jest szablon bazowy.

Ten arkusz stylów definiuje wygląd i sposób działania naszych wszystkich stron internetowych.

Te dyrektywy silnika Jinja2 wskazują, że przed renderowaniem strony zostanie tu podstawiony blok kodu HTML i że zostanie on dostarczony przez każdą stronę, która po tej dziedziczy.

Gdy mamy już gotowy szablon bazowy, możemy po nim dziedziczyć, korzystając z dyrektywy `extends` zapewnianej przez silnik Jinja2. Gdy skorzystamy z tej możliwości, dziedziczące pliki HTML będą musiały dostarczyć tylko kod HTML wymagany przez poszczególne bloki nazwane, które znajdują się w naszym szablonie bazowym. W przypadku naszego szablonu jest tam tylko jeden blok: `body`.

Poniżej znajduje się kod HTML pierwszej z naszych stron, którą zapisaliśmy w pliku *entry.html*. Tworzy on formularz HTML, z którego użytkownicy mogą korzystać, aby wprowadzać wartości zmiennych `phrase` oraz `letters`, których otrzymania spodziewa się nasza aplikacja.

Zwróć uwagę, że udało się nam uniknąć w tym pliku powtarzania „standardowego” kodu HTML znajdującego się już w szablonie bazowym, ponieważ za uwzględnienie go odpowiada dyrektywa `extends`. Musimy tu jedynie dostarczyć treści HTML, które są specyficzne dla tego pliku. Robi się to, podając odpowiedni kod w obrębie bloku silnika Jinja2 o nazwie `body`, co pokazano poniżej.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<form method='POST' action='/search4'>
<table>
<p> Użyj tego formularza, aby określić parametry wyszukiwania:</p>
<tr><td>Fraza:</td><td><input name='phrase' type='TEXT' width='60'></td></tr>
<tr><td>Litery:</td><td><input name='letters' type='TEXT' value='aeiou'></td></tr>
</table>
<p> Gdy będziesz gotów, kliknij ten przycisk:</p>
<p><input value='Zrób to!' type='SUBMIT'></p>
</form>

{% endblock %}
```

Ten szablon dziedziczy po szablonie bazowym i dostarcza treść, która powinna zostać wstawiona w miejscu bloku o nazwie „body”.

I wreszcie: poniżej znajdziesz kod HTML tworzący plik *results.html*, który wykorzystujemy, aby wyświetlić wyniki naszego wyszukiwania. Również ten szablon dziedziczy po szablonie bazowym.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>Podałeś następujące dane:</p>
<table>
<tr><td>Fraza:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Litery:</td><td>{{ the_letters }}</td></tr>
</table>

<p>Gdy we frazie "{{ the_phrase }}" wyszuka się litery "{{ the_letters }}", zwrócone zostaną
następujące wyniki:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

Podobnie jak w przypadku pliku entry.html, również ten szablon dziedziczy po szablonie bazowym i również on dostarcza treść, która powinna zostać wstawiona w miejscu bloku o nazwie „body”.

Zwróć uwagę na te dodatkowe wartości argumentów, które są Ci potrzebne, aby dostarczyć odpowiednie wartości przed renderowaniem strony.

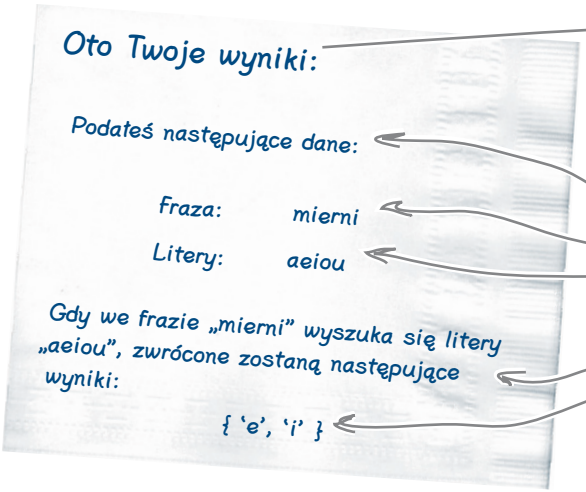
Szablony mają związek ze stronami WWW

Nasza aplikacja WWW powinna renderować dwie strony WWW i mamy teraz dwa szablony, które nam w tym pomogą. Obydwa szablony dziedziczą po szablonie bazowym, otrzymując w ten sposób wygląd i sposób działania właściwy dla tego szablonu. Wszystko, co musimy teraz zrobić, ogranicza się do wyrenderowania naszych stron.

Pobierz te szablony (i arkusze stylów) spod adresu *ftp://ftp.helion.pl/przyklady/pytrg2.zip*.

Zanim zobaczymy, jak framework Flask (wraz z silnikiem Jinja2) renderuje strony, spójrzmy raz jeszcze na nasze „serwetkowe specyfikacje" zestawione z kodem tworzącym odpowiednie szablony. Zauważ, jak kod HTML umieszczony wewnątrz dyrektywy `{% block %}` Jinja2 dokładnie pasuje do naszkicowanego ręcznie projektu strony. Brakuje tu już tylko właściwie tytułu strony, który pojawi się podczas renderowania w miejscu zajmowanym przez dyrektywę `{{ the_title }}`. Traktuj każdą nazwę umieszczoną w podwójnych nawiasach klamrowych jak argument szablonu.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }} </h2>

<form method='POST' action='/search4'>
<table>
<p> Użyj tego formularza, aby określić parametry wyszukiwania:</p>
<tr><td>Fraza:</td><td><input name='phrase' type='TEXT'
width='60'></td></tr>
<tr><td>Litery:</td><td><input name='letters' type='TEXT'
value='aeiou'></td></tr>
</table>
<p> Gdy będziesz gotów, kliknij ten przycisk:</p>
<p><input value='Zrób to!' type='SUBMIT'></p>
</form>

{% endblock %}
```




```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }} </h2>

<p>Podałeś następujące dane:</p>
<table>
<tr><td>Fraza:</td><td>{{ the_phrase }} </td></tr>
<tr><td>Litery:</td><td>{{ the_letters }} </td></tr>
</table>

<p>Gdy we frazie "{{ the_phrase }}" wyszuka się litery "{{ the_
letters }}", zwrócone zostaną następujące wyniki:</p>
<h3>{{ the_results }} </h3>

{% endblock %}
```

Nie zapomnij o tych dodatkowych argumentach.

Renderowanie szablonów z poziomu Flaska

Flask zapewnia funkcję o nazwie `render_template`, która zwraca łańcuch znakowy zawierający kod HTML, gdy poda jej się nazwę szablonu oraz wszystkie wymagane argumenty. Aby skorzystać z funkcji `render_template`, dodaj jej nazwę do listy elementów importowanych z modułu `flask` (znajdującej się na początku Twojego kodu), a następnie wywołaj ją, gdy będzie to konieczne.

Zanim to jednak zrobimy, zmieńmy nazwę pliku zawierającego kod naszej aplikacji WWW (teraz brzmi ona *hello_flask.py*) na bardziej odpowiednią. Możesz tu użyć dowolnej nazwy, jaka tylko przyjdzie Ci do głowy, ale my proponujemy nazwę *vsearch4web.py*. Poniżej znajdziesz kod, który znajduje się obecnie w tym pliku.

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

@app.route('/search4')
def do_search() -> str:
   return str(search4letters('życie, wszechświat i cała reszta', 'eiru,!'))

app.run()
```

Ten kod znajduje się obecnie w pliku o nazwie „vsearch4web.py".

Aby wyrenderować ten formularz HTML na bazie szablonu *entry.html*, w przedstawionym powyżej kodzie musimy wprowadzić kilka istotnych zmian. Oto co powinieneś zrobić:

1. **Zaimportuj funkcję `render_template`.**
   Dodaj funkcję `render_template` do listy importowanych elementów w wierszu zaczynającym się od konstrukcji `from flask` i znajdującym się na początku kodu.

2. **Utwórz nowy URL; w tym przypadku: `/entry`.**
   Zawsze gdy potrzebujesz nowego ciągu URL w swojej aplikacji WWW powstałej na bazie frameworka Flask, powinieneś dodać nowy wiersz zawierający konstrukcję `@app.route`. Dołożymy go przed wierszem kodu zawierającym wywołanie metody `app.run()`.

3. **Utwórz funkcję, która zwraca poprawnie wyrenderowany kod HTML.**
   Jeśli masz już napisany wiersz zawierający dekorator `@app.route`, możesz powiązać z nim kod, tworząc funkcję, która wykonuje rzeczywiste działanie (i sprawia, że Twoja aplikacja jest bardziej przydatna dla użytkowników). Funkcja ta wywołuje funkcję `render_template` (i zwraca otrzymany od niej wynik), przekazując jej nazwę pliku szablonu (w tym przypadku *entry.html*), jak również wszelkie wartości argumentów, które są wymagane przez ten szablon (w tym przypadku potrzebna nam jest wartość argumentu `the_title`).

Wprowadźmy zmiany w naszym istniejącym kodzie.

Wyświetlanie formularza HTML aplikacji WWW

Wprowadźmy zmiany opisane szczegółowo na końcu poprzedniej strony, dodając odpowiedni kod do naszej aplikacji. Wprowadź w swoim kodzie takie same modyfikacje jak te, które zostały przedstawione poniżej.

1. **Zaimportuj funkcję** `render_template`.

```
from flask import Flask, render_template
```

Dodaj funkcję render_template do listy elementów importowanych z modułu flask.

2. **Utwórz nowy URL; w tym przypadku:** `/entry`.

```
@app.route('/entry')
```

Pod wierszem zawierającym funkcję do_search, ale przed wierszem zawierającym wywołanie app.run() wstaw wiersz, za pomocą którego dodasz do aplikacji WWW nowy ciąg URL.

3. **Utwórz funkcję, która zwraca poprawnie wyrenderowany kod HTML.**

```
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Witamy na stronie internetowej search4letters!')
```

Podaj nazwę szablonu do wyrenderowania.

Dodaj tę funkcję bezpośrednio pod wierszem nowego dekoratora @app.route.

Podaj wartość, która ma być powiązana z argumentem the_title.

Po wprowadzeniu wymienionych powyżej zmian kod naszej aplikacji WWW wygląda tak, jak zostało to pokazane poniżej (dodane wiersze zostały wyróżnione).

```
from flask import Flask, render_template
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

@app.route('/search4')
def do_search() -> str:
   return str(search4letters('życie, wszechświat i cała reszta', 'eiru,!'))

@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

app.run()
```

Na razie pozostawiamy resztę tego kodu bez zmian.

Przygotowanie do uruchomienia kodu szablonu

Aż prosi się, aby otworzyć wiersz poleceń, a następnie uruchomić najnowszą wersję naszego kodu. Jednak z wielu różnych powodów aplikacja nie zacznie od razu działać.

Jeśli tego jeszcze nie zrobiłeś, pobierz szablony i arkusze stylów spod adresu *ftp://ftp.helion.pl/przyklady/pytrg2.zip*.

Szablon bazowy odwołuje się do arkusza stylów *hf.css*, a plik ten powinien się znajdować w folderze o nazwie *static* (podrzędnym wobec folderu zawierającego Twój kod). Poniżej znajdziesz fragment szablonu bazowego, w którym występuje to odwołanie.

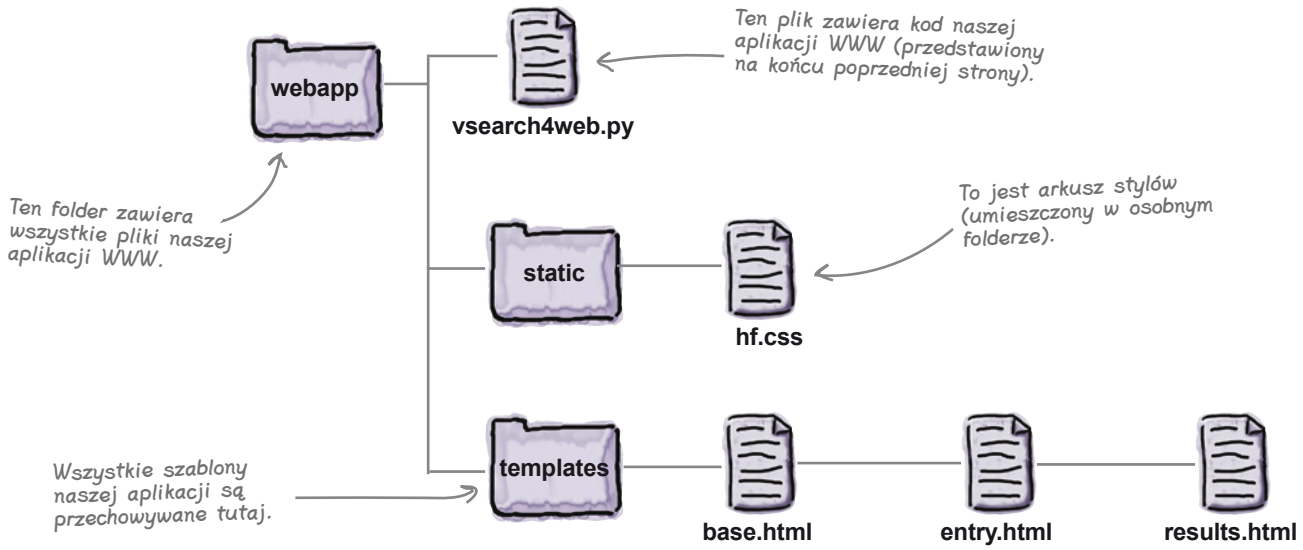
```
...
    <title>{{ the_title }}</title>
    <link rel="stylesheet" href="static/hf.css" />
</head>
...
```

Plik hf.css powinien istnieć (i znajdować się w folderze static).

Nie krępuj się i korzystaj z kopii pliku CSS wchodzącego w skład archiwum kodów związanych z niniejszą książką (adres, pod którym znajdziesz to archiwum, został podany na marginesie tej strony). Pamiętaj tylko, aby pobrany arkusz stylów umieścić w folderze o nazwie *static*.

Poza tym framework Flask wymaga, aby Twoje szablony były przechowywane w folderze o nazwie *templates*, który — podobnie jak folder *static* — powinien być folderem podrzędnym wobec folderu zawierającego Twój kod. Wśród plików związanych z tym rozdziałem znajdziesz również wszystkie niezbędne szablony, dzięki czemu możesz oszczędzić sobie przepisywania całego tego kodu HTML!

Przy założeniu, że plik z kodem Twojej aplikacji WWW umieszczony jest w folderze *webapp*, poniżej możesz zobaczyć strukturę folderów, która jest Ci potrzebna do uruchomienia najnowszej wersji programu *vsearch4web.py*.



Jesteśmy gotowi, aby testowo uruchomić aplikację

Jeśli przygotowałeś już wszystko, co trzeba — tzn. pobrałeś arkusz stylów i szablony, a także zaktualizowałeś swój kod — jesteś gotów, aby ponownie wziąć w obroty swoją aplikację WWW wykorzystującą framework Flask.

Poprzednia wersja kodu prawdopodobnie nadal działa w wierszu poleceń Twojego systemu operacyjnego.

Wróć teraz do tego okna i użyj skrótu klawiszowego *Ctrl+C*, aby zatrzymać poprzednią wersję aplikacji. Następnie naciśnij klawisz *strzałki skierowanej w górę*, aby przywołać ostatnie polecenie, zmień nazwę pliku, który ma zostać uruchomiony, a na koniec naciśnij klawisz *Enter*. W wyniku tego powinna zostać uruchomiona nowa wersja Twojego kodu, a na ekranie powinny pojawić się standardowe komunikaty stanu.

```
...
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Jun/2017 23:23:19] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Jun/2017 23:23:29] "GET /search4 HTTP/1.1" 200 -
^C
C:\Users\lsuma\Desktop\Python\R5\webapp>py -3 vsearch4web.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Uruchom swój nowy kod (znajdujący się w pliku vsearch4web.py).

Ponownie zatrzymaj aplikację WWW...

Nowy kod wystartował i działa; aplikacja oczekuje teraz na żądania do obsłużenia.

Przypomnij sobie, że nowa wersja aplikacji w dalszym ciągu obsługuje URL-e / oraz `/search4`, dlatego gdy korzystając z przeglądarki internetowej, zażądasz któregoś z nich, odpowiedzi będą takie same jak te, które zostały przedstawione we wcześniejszej części tego rozdziału. Jeśli jednak użyjesz tego URL-a:

http://127.0.0.1:5000/entry

w Twojej przeglądarce internetowej pojawi się odpowiedź w postaci wyrenderowanego formularza HTML (który został pokazany na początku kolejnej strony). W wierszu poleceń powinny zostać wyświetlone dwa dodatkowe wiersze stanu: jeden związany z żądaniem `/entry` oraz drugi związany z żądaniem Twojej przeglądarki dotyczącym arkusza stylów *hf.css*.

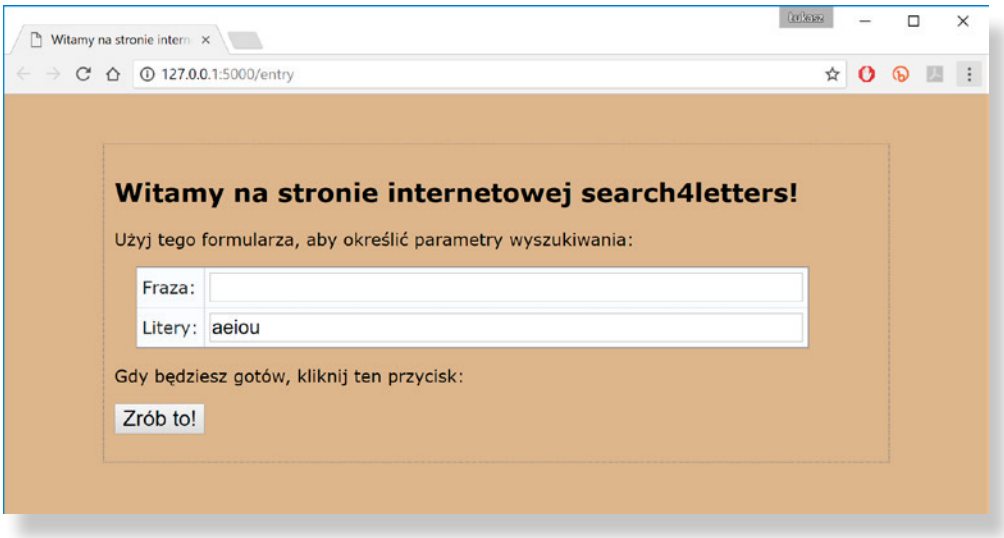
Ty żądasz formularza HTML...

...a Twoja przeglądarka żąda arkusza stylów.

```
...
127.0.0.1 - - [14/Jun/2017 21:53:47] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [14/Jun/2017 21:53:47] "GET /static/hf.css HTTP/1.1" 200 -
```

Jazda próbna

Poniżej możesz zobaczyć, co pojawi się w oknie przeglądarki internetowej, gdy wprowadzisz w jej polu adresu URL **`http://127.0.0.1:5000/entry`** i naciśniesz klawisz *Enter*.



Wygląda dobrze.

Raczej nie uda nam się zdobyć za tę stronę żadnej nagrody dla najlepszych projektów WWW, ale wygląda ona poprawnie i przypomina to, co naszkicowaliśmy wcześniej na naszej serwetce. Niestety, gdy wprowadzisz frazę i (opcjonalnie) zmienisz zestaw liter do wyszukania, a następnie klikniesz przycisk *Zrób to!*, ujrzysz stronę błędu, która przedstawiona została poniżej.

Ups! To nie może być nic dobrego.

Lekka wpadka, nie sądzisz? Zobaczmy, co się tu dzieje.

Znaczenie kodów stanu HTTP

Gdy z Twoją aplikacją WWW dzieje się coś złego, serwer WWW odpowiada za pomocą kodu stanu HTTP (który jest przesyłany do Twojej przeglądarki). HTTP to protokół komunikacji, który umożliwia przeglądarkom internetowym i serwerom komunikowanie się ze sobą. Znaczenie poszczególnych kodów stanu jest ściśle określone (więcej informacji na ten temat znajdziesz w widocznej obok ramce „Porady maniaka"). W gruncie rzeczy każde *żądanie* sieciowe generuje *odpowiedź* w postaci kodu stanu HTTP.

Aby sprawdzić kod stanu, który został wysłany do Twojej przeglądarki internetowej przez Twoją aplikację WWW, przejrzyj komunikaty stanu widoczne w oknie wiersza poleceń. Poniżej możesz zobaczyć, co nam udało się w nim znaleźć.

```
    ...
127.0.0.1 - - [14/Jun/2017 21:53:47] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [14/Jun/2017 22:01:11] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [14/Jun/2017 22:06:30] "POST /search4 HTTP/1.1" 405 -
```

Aha. Coś poszło nie tak, a serwer wygenerował kod stanu oznaczający błąd po stronie klienta.

Kod stanu 405 wskazuje, że klient (Twoja przeglądarka) wysłał żądanie, korzystając z metody HTTP, której serwer nie dopuszcza. Dostępnych jest sporo różnych metod HTTP, ale dla nas w tym momencie istotne są właściwie tylko dwie: `GET` oraz `POST`.

1. **Metoda `GET`.**
   Przeglądarka korzysta z tej metody przeważnie po to, aby zażądać od serwera WWW jakiegoś zasobu; jest to zdecydowanie najczęściej używana metoda. (Piszemy tu „przeważnie", ponieważ można korzystać z metody `GET`, aby — co dość mylące — *wysyłać* dane z przeglądarki do serwera, ale tą możliwością nie będziemy się teraz zajmować). Wszystkie URL-e w naszej aplikacji WWW obsługują w tej chwili metodę `GET`, która jest standardową metodą HTTP dla Flaska.

2. **Metoda `POST`.**
   Ta metoda pozwala przeglądarce wysyłać dane do serwera za pośrednictwem HTTP i jest ściśle związana ze znacznikiem `<form>` języka HTML. Możesz kazać swojej aplikacji WWW korzystającej z Flaska akceptować dane pochodzące z przeglądarki przez dostarczenie dekoratorowi `@app.route` dodatkowego argumentu.

Zmieńmy wiersz zawierający dekorator `@app.route` związany z URL-em `/search` naszej aplikacji w taki sposób, aby akceptowała ona dane przekazywane za pomocą metody `POST`. Aby to zrobić, wróć do okna edytora i ponownie zmodyfikuj kod znajdujący się w pliku *vsearch4web.py*.

Porady maniaka

Poniżej znajdziesz krótkie i niedokładne wyjaśnienie znaczenia różnych kodów stanu HTTP, które mogą być wysyłane przez serwer sieciowy (na przykład Twoją aplikację WWW korzystającą z Flaska) do klienta WWW (na przykład Twojej przeglądarki internetowej).

Istnieje pięć głównych kategorii kodów stanu: setki (1xx), dwusetki (2xx), trzysetki (3xx), czterysetki (4xx) i pięćsetki (5xx).

Kody należące do przedziału od **100** do **199** to komunikaty **informacyjne**, mówiące, że wszystko jest w porządku i zawierające przekazywane przez serwer szczegóły związane z żądaniem klienta.

Kody należące do przedziału od **200** do **299** stanowią komunikaty **powodzenia**, informujące, że serwer otrzymał, zrozumiał i przetworzył żądanie klienta. Wszystko jest zatem w porządku.

Kody należące do przedziału od **300** do **399** to komunikaty **przekierowania**, za pomocą których serwer informuje klienta, że jego żądanie może być obsłużone gdzieś indziej.

Kody należące do przedziału od **400** do **499** stanowią komunikaty **błędu klienta**, informujące, że serwer otrzymał od klienta żądanie, którego nie rozumie i nie może przetworzyć. Winę ponosi tu zwykle klient.

Kody należące do przedziału od **500** do **599** to komunikaty **błędu serwera**, mówiące, że serwer otrzymał od klienta żądanie, ale nie udało mu się go przetworzyć. Winę ponosi tu zwykle serwer.

Więcej szczegółów na temat kodów odpowiedzi HTTP znajdziesz pod adresem *https://pl.wikipedia.org/wiki/Kod_odpowiedzi_HTTP*.

Obsługa danych przekazywanych za pomocą metody POST

Oprócz przyjmowania ciągu URL w roli pierwszego argumentu dekorator `@app.route` akceptuje też inne, opcjonalne argumenty.

Jednym z nich jest argument `methods`, który określa listę metod HTTP obsługiwanych przez dany URL. Flask obsługuje domyślnie metodę `GET` w przypadku wszystkich URL-i. Jeśli jednak argumentowi `methods` przypisze się jakąś wartość, standardowe zachowanie zostaje zastąpione zdefiniowanym przez ten argument. Poniżej możesz zobaczyć, jak w tej chwili wygląda wiersz zawierający nasz dekorator `@app.route`.

```
@app.route('/search4')
```

Nie określiliśmy tu metody HTTP, która powinna być obsługiwana, dlatego Flask standardowo założył obsługę metody GET.

Jeśli chcesz sprawić, aby URL `/search4` obsługiwał metodę `POST`, dodaj do dekoratora argument `methods` i przypisz do niego listę metod HTTP, które powinien obsługiwać Twój URL. Przedstawiony poniżej wiersz kodu określa, że URL `/search4` obsługuje teraz wyłącznie metodę `POST` (co oznacza, że żądania `GET` nie są już przezeń obsługiwane).

```
@app.route('/search4', methods=['POST'])
```

URL /search4 zapewnia teraz obsługę jedynie metody POST.

Ta niewielka zmiana to już wszystko, co jest nam potrzebne, aby pozbyć się komunikatu „Method Not Allowed" (niedozwolona metoda), ponieważ metoda `POST` formularza HTML pasuje do metody `POST` zdefiniowanej w wierszu dekoratora `@app.route`.

Ten fragment kodu HTML pochodzi z pliku entry.html...

```
...
<form method='POST' action='/search4'>
<table>
...
```

...a ten kod Pythona wzięto z pliku vsearch4web.py.

```
...
@app.route('/search4', methods=['POST'])
def do_search() -> str:
...
```

Zwróć uwagę, że w HTML-u używa się słowa „method" (w liczbie pojedynczej), podczas gdy Flask korzysta ze słowa „methods" (w liczbie mnogiej).

Nie istnieją głupie pytania

P: A co, gdy mój URL powinien obsługiwać zarówno metodę `GET`, jak i `POST`? Czy to w ogóle możliwe?

O: Tak, musisz tylko dodać nazwę tej metody HTTP, która jest Ci potrzebna, do listy przypisywanej argumentowi `methods`. Gdybyś na przykład chciał, aby URL `/search4` obsługiwał również metodę `GET`, powinieneś jedynie zmienić swój dekorator `@app.route` w taki sposób, aby przybrał następującą postać: `@app.route('/search4', methods=['GET', 'POST'])`. Więcej informacji na ten temat znajdziesz w dokumentacji Flaska umieszczonej pod adresem *http://flask.pocoo.org*.

Ulepszenie cyklu edycji, zatrzymywania, uruchamiania i testowania

Na tym etapie, gdy zapisałeś już swój zmodyfikowany kod, dobrze byłoby zatrzymać aplikację WWW działającą w wierszu poleceń, a następnie uruchomić ją ponownie, aby przetestować nowe zmiany. Używany przez nas tutaj cykl edycji, zatrzymywania, uruchamiania i testowania działa, ale po pewnym czasie staje się dość nużący (zwłaszcza gdy masz do czynienia z długą serią drobnych poprawek w kodzie swojej aplikacji WWW).

Aby zwiększyć wydajność tego procesu, framework Flask umożliwia Ci uruchamianie aplikacji WWW w *trybie debugowania*, który (oprócz innych rzeczy) automatycznie restartuje Twoją aplikację zawsze wtedy, gdy wykryta zostanie zmiana w jej kodzie (zwykle w wyniku wprowadzenia modyfikacji i zapisania pliku). Warto korzystać z tej możliwości, dlatego włączmy tryb debugowania, zmieniając ostatni wiersz kodu w pliku *vsearch4web.py* w taki sposób, aby wyglądał jak poniżej.

```
app.run(debug=True)
```

Włączenie debugowania

Kod Twojego programu powinien teraz prezentować się tak, jak zostało to pokazane poniżej.

```
from flask import Flask, render_template
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> str:
   return str(search4letters('życie, wszechświat i cała reszta', 'eiru,!'))

@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')
app.run(debug=True)
```

Teraz jesteśmy już gotowi do testowego uruchomienia tego kodu. Aby to zrobić, zatrzymaj działającą w tej chwili aplikację WWW (już po raz ostatni), używając skrótu klawiszowego *Ctrl+C*, a następnie ponownie ją uruchom, naciskając klawisz *strzałki skierowanej w górę*, a potem klawisz *Enter*.

Tym razem zamiast pokazywać jedynie zwykły komunikat zaczynający się od słów `Running on http://127...`, Flask wyświetlił trzy nowe wiersze stanu będące swego rodzaju potwierdzeniem, że tryb debugowania jest aktywny. Poniżej możesz zobaczyć, jak wyglądały one na naszym komputerze.

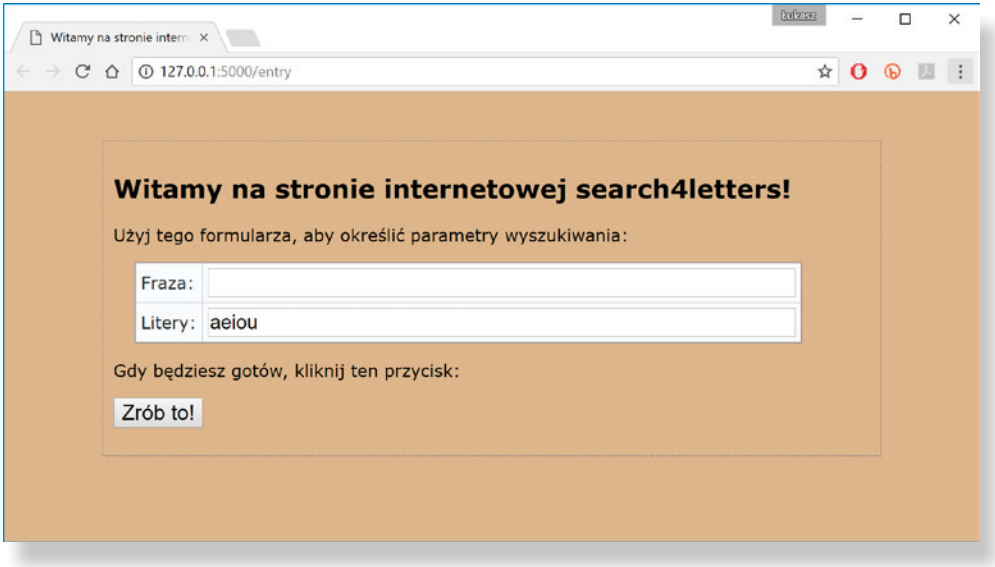
```
C:\Users\lsuma\Desktop\Python\R05\webapp>py -3 vsearch4web.py
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 103-121-075
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

To sposób wykorzystywany przez framework Flask, aby poinformować Cię, że Twoja aplikacja WWW zostanie automatycznie zrestartowana, gdy zmieni się jej kod. Nie martw się też, jeśli Twój PIN debugowania jest inny niż nasz (to zupełnie w porządku). I tak nie będziemy go teraz używać.

Teraz, gdy udało nam się już uruchomić naszą najnowszą wersję aplikacji, spróbujmy jej trochę poużywać, aby przekonać się, co się zmieniło.

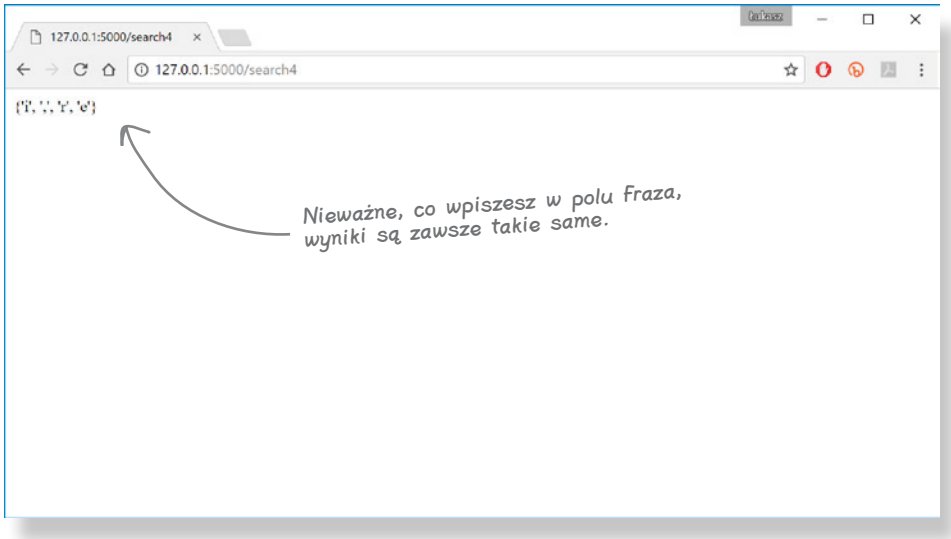
Jazda próbna

Wróć do formularza wprowadzania danych — podaj w polu adresu swojej przeglądarki internetowej URL **`http://127.0.0.1:5000/entry`** i naciśnij klawisz *Enter*.



Strona nadal wygląda dobrze.

Błąd niedozwolonej metody („Method Not Allowed") zniknął, ale aplikacja nadal nie działa prawidłowo. Możesz co prawda wpisać w tym formularzu dowolną frazę, a następnie kliknąć przycisk *Zrób to!* i nie pojawi się komunikat błędu, lecz jeśli spróbujesz zrobić to kilka razy, zauważysz, że wyniki będą zawsze takie same (niezależnie od danych wprowadzonych w polach *Fraza* i *Litery*), tak jak zostało to pokazane poniżej. Sprawdźmy, jaka jest tego przyczyna.



Dostęp do danych formularza HTML za pomocą Flaska

Nasza aplikacja nie zgłasza już błędu niedozwolonej metody („Method Not Allowed"). Zamiast tego mamy inny problem, polegający na tym, że zawsze zwraca ona ten sam zestaw znaków: *i*, *przecinek*, *r* oraz *e*. Jeśli tylko spojrzysz w kod, który jest wykonywany, gdy odwołujesz się do URL-a `/search4`, szybko odkryjesz powód tego dziwnego zachowania. Jest to spowodowane tym, że wartości argumentów `phrase` i `letters` są *zapisane na sztywno* (ang. *hardcoded*) w wywołaniu funkcji `search4letters`.

```
        ...
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    return str(search4letters('życie, wszechświat i cała reszta', 'eiru,!'))
        ...
```

Nieważne, co wpiszemy w formularzu HTML, nasz kod zawsze będzie używał tych zapisanych na sztywno wartości.

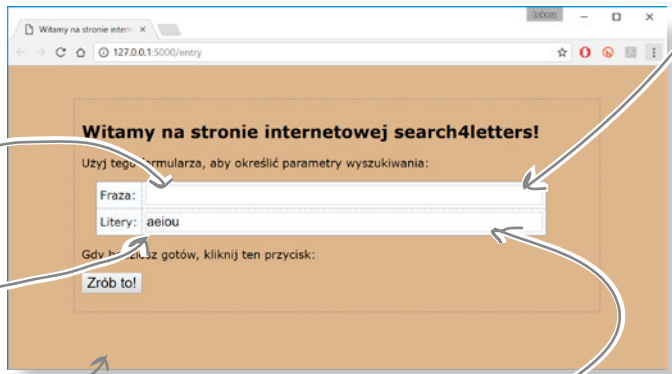
Nasz formularz HTML przesyła swoje dane do serwera WWW, ale aplikacja po prostu je ignoruje. Aby zrobić z nich użytek, powinniśmy zmienić kod naszej aplikacji WWW w taki sposób, aby przyjmowała dane, a następnie przeprowadzała na nich jakąś operację.

Framework Flask oferuje wbudowany obiekt o nazwie `request`, który zapewnia dostęp do przesłanych danych. Obiekt ten zawiera słownikowy atrybut o nazwie `form`, który oferuje z kolei dostęp do danych formularza HTML przekazanych przez przeglądarkę. Jako że `form` jest zwykłym słownikiem Pythona, obsługuje standardową notację nawiasów kwadratowych, którą poznałeś w rozdziale 3. Aby odwołać się do jakiejś danej pochodzącej z formularza, po prostu umieść nazwę odpowiedniego elementu w nawiasach kwadratowych.

Dane z tego elementu formularza są dostępne w kodzie naszej aplikacji WWW jako `request.form['phrase']`.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }} </h2>

<form method='POST' action='/search4'>
<table>
<p> Użyj tego formularza, aby określić parametry wyszukiwania:</p>
<tr><td>Fraza:</td><td><input name='phrase' type='TEXT'
width='60'></td></tr>
<tr><td>Litery:</td><td><input name='letters' type='TEXT'
value='aeiou'></td></tr>
</table>
<p> Gdy będziesz gotów, kliknij ten przycisk:</p>
<p><input value='Zrób to!' type='SUBMIT'></p>
</form>
{% endblock %}
```

Szablon HTML (zapisany w pliku entry.html)



Wyrenderowany formularz w naszej przeglądarce internetowej

Dane z tego elementu formularza są dostępne w kodzie naszej aplikacji WWW jako `request.form['letters']`.

Używanie danych żądania w swojej aplikacji WWW

Aby skorzystać z obiektu `request`, zaimportuj go w wierszu zaczynającym się od słów `from flask` i znajdującym się na początku kodu Twojego programu, a następnie odwołaj się do odpowiedniego elementu za pomocą konstrukcji `request.form` zgodnie z potrzebą. W tym przypadku chodzi o zastąpienie wartości zapisanych na sztywno w funkcji `do_search` danymi pochodzącymi z naszego formularza HTML. Dzięki temu za każdym razem, gdy zostaną w nim zatwierdzone wartości `phrase` i `letters`, nasza aplikacja zwróci odpowiednie dla nich wyniki.

Dokonajmy odpowiednich zmian w kodzie programu. Zacznijmy od dodania obiektu `request` do listy elementów importowanych z modułu `flask`. Aby to zrobić, zmień pierwszy wiersz kodu w pliku *vsearch4web.py* tak, aby wyglądał jak poniżej.

```
from flask import Flask, render_template, request
```

Dodaj obiekt request do listy importowanych elementów.

Dzięki informacjom zaprezentowanym na poprzedniej stronie wiemy już, że do danej `phrase` wprowadzonej w formularzu HTML możemy się odwołać z poziomu naszego kodu, korzystając z konstrukcji `request.form['phrase']`, a dostęp do wartości `letters` zapewni nam konstrukcja `request.form['letters']`. Zmodyfikujmy funkcję `do_search` w taki sposób, aby korzystała z tych wartości (i usuńmy zapisane na sztywno łańcuchy znakowe).

Utwórz dwie nowe zmienne...

```
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    phrase = request.form['phrase']
    letters = request.form['letters']
    return str(search4letters(phrase, letters))
```

i przypisz do nich odpowiednie dane pochodzące z formularza HTML...

...a następnie użyj tych zmiennych w wywołaniu funkcji search4letters.

Automatyczne przeładowania

Zanim zrobisz cokolwiek innego (ale już po wykonaniu przedstawionych powyżej zmian w swoim kodzie), zapisz swój plik *vsearch4web.py*, a następnie przejdź do swojego wiersza poleceń i spójrz na komunikaty stanu wygenerowane przez Twoją aplikację WWW. Poniżej możesz się przekonać, co my tam zobaczyliśmy (u Ciebie powinno to wyglądać podobnie).

```
$ python3 vsearch4web.py
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
127.0.0.1 - - [23/Nov/2015 22:39:11] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 22:39:11] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 22:17:58] "POST /search4 HTTP/1.1" 200 -
 * Detected change in 'vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
```

Debugger frameworka Flask wykrył zmiany w kodzie i zrestartował Twoją aplikację WWW za Ciebie. Przyznasz chyba, że to całkiem wygodne?

Nie wpadaj w panikę, jeśli w swoim wierszu poleceń ujrzysz coś innego. Automatyczne przeładowanie działa wyłącznie wtedy, gdy wprowadzone przez Ciebie zmiany są prawidłowe. Jeśli w Twoim kodzie pojawią się jakieś błędy, aplikacja WWW zacznie bombardować wiersz poleceń komunikatami błędów. Aby ruszyć dalej, popraw błędy występujące w Twoim kodzie i zrestartuj swoją aplikację ręcznie (naciskając klawisz *strzałki skierowanej w górę*, a potem klawisz *Enter*).

Jazda próbna

Teraz, gdy zmieniliśmy naszą aplikację WWW w taki sposób, aby przyjmowała (i przetwarzała) dane pochodzące z formularza HTML, możemy wprowadzić do niego różne frazy i litery, a aplikacja powinna robić to, czego od niej oczekujemy.

Witamy na stronie interneto

Użyj tego formularza, aby określić parametry w

Fraza: To jest test możliwości przesyłania

Litery: aeiou

Gdy będziesz gotów, kliknij ten przycisk:

Zrób to!

127.0.0.1:5000/search4

{'e', 'o', 'a', 'i'}

Fraza zawiera wszystkie litery przekazane serwerowi WWW.

Witamy na stronie interne

Użyj tego formularza, aby określić paramet

Fraza: życie, wszechświat i cała reszta

Litery: vxy

Gdy będziesz gotów, kliknij ten przycis

Zrób to!

127.0.0.1:5000/search4

{'y'}

W przekazanej frazie występuje tylko litera „y".

Witamy na stronie inte

Użyj tego formularza, aby określić par

Fraza: autostopowicz

Litery: edfkm

Gdy będziesz gotów, kliknij ten przyc

Zrób to!

127.0.0.1:5000/search4

set()

Pamiętaj: pusty zbiór jest wyświetlany jako „set()", co oznacza, że żadna z liter „e", „d", „f", „k", „m" nie została znaleziona w podanej frazie.

Generowanie wyników w postaci kodu HTML

Na tym etapie nasza aplikacja WWW spełnia już swoją funkcję. Za pomocą dowolnej przeglądarki internetowej można podać kombinację frazy i liter, a aplikacja wywoła w naszym imieniu metodę `search4letters` i wyświetli nam zwrócone przez nią wyniki. Jednak dane wyjściowe naszej aplikacji nie są tak naprawdę stroną w HTML-u, lecz stanowią jedynie surowe wartości przekazane w postaci tekstowej do oczekującej ich przeglądarki (która wyświetla je w swoim oknie).

Przypomnij sobie nasze „serwetkowe specyfikacje", które zostały zaprezentowane we wcześniejszej części tego rozdziału. Poniżej możesz zobaczyć, co chcieliśmy osiągnąć.

Ta część jest zrobiona. Dzięki szablonowi entry.html udało nam się wygenerować formularz przypominający ten projekt.

Witamy na stronie internetowej search4letters!

Użyj tego formularza, aby określić parametry wyszukiwania:

Fraza:

Litery: aeiou

Gdy będziesz gotów, kliknij ten przycisk:

Zrób to!

Oto Twoje wyniki:

Podałeś następujące dane:

Fraza: mierni

Litery: aeiou

Gdy we frazie „mierni" wyszuka się litery „aeiou", zwrócone zostaną następujące wyniki:

{ 'e', 'i' }

Ta część została nam do zrobienia. W tej chwili wyświetlamy jedynie wyniki w postaci surowych danych.

Gdy była mowa o możliwościach silnika szablonów Jinja2, przedstawiliśmy Ci dwa szablony HTML. Pierwszy z nich, *entry.html*, jest używany do generowania formularza. Drugi, *results.html*, miał zostać wykorzystany do wyświetlania wyników działania aplikacji. Użyjmy go teraz, aby nasze surowe dane przekształcić w stronę w HTML-u.

Nie istnieją głupie pytania

P: Czy da się skorzystać z możliwości silnika Jinja2 do tworzenia szablonów danych tekstowych innego rodzaju niż strony w HTML-u?

O: Tak, Jinja2 to silnik szablonów tekstowych, którego można używać na wiele różnych sposobów. Co prawda znajduje on zwykle zastosowanie w projektach stron WWW (jak miało to miejsce tutaj w połączeniu z frameworkiem Flask), ale nie ma żadnych przeciwwskazań, abyś korzystał z niego w stosunku do danych tekstowych, jeśli tylko masz taką potrzebę.

Obliczanie danych, które są nam potrzebne

Przypomnijmy sobie zawartość pliku szablonu *results.html*, który został zaprezentowany we wcześniejszej części tego rozdziału. Znaczniki charakterystyczne dla silnika Jinja2 zostały poniżej wyróżnione.

To jest plik results.html.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>Podałeś następujące dane:</p>
<table>
<tr><td>Fraza:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Litery:</td><td>{{ the_letters }}</td></tr>
</table>

<p>Gdy we frazie "{{ the_phrase }}" wyszuka się litery "{{ the_letters }}", zwrócone zostaną
następujące wyniki:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

Wyróżnione nazwy umieszczone w podwójnych nawiasach klamrowych to zmienne używane przez silnik Jinja2, które przyjmują wartości określone przez odpowiadające im zmienne występujące w Twoim kodzie Pythona. W naszym przypadku mamy do czynienia z czterema takimi zmiennymi. Są to: `the_title`, `the_phrase`, `the_letters` oraz `the_results`. Spójrz raz jeszcze na (przedstawiony poniżej) kod funkcji `do_search`, który niebawem zmienimy w taki sposób, aby renderowała ona szablon HTML przedstawiony powyżej. Jak mogłeś już zauważyć, w kodzie tym pojawiają się dwie z czterech zmiennych, które są nam potrzebne do wyrenderowania naszego szablonu (aby maksymalnie uprościć sobie życie, w kodzie Pythona użyliśmy nazw zmiennych podobnych do tych, które są wykorzystywane w szablonie Jinja2).

Tu znajdują się dwie z czterech zmiennych, które są nam potrzebne.

```
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    phrase = request.form['phrase']
    letters = request.form['letters']
    return str(search4letters(phrase, letters))
```

Dwa pozostałe wymagane argumenty szablonu (`the_title` oraz `the_results`) nadal trzeba utworzyć, korzystając ze zmiennych używanych w tej funkcji oraz przypisywanych do nich wartości.

Do zmiennej `the_title` możemy przypisać łańcuch znakowy `"Oto Twoje wyniki:"`, a następnie do zmiennej `thre_results` przypisać wywołanie funkcji `search4letters`. Dzięki temu wszystkie cztery niezbędne wartości będzie można przekazać szablonowi *results.html* jako argumenty przed wyrenderowaniem strony.

Magnesiki szablonów

Autorzy serii *Rusz głową!* zebrali się razem i na podstawie wymagań dotyczących aktualizacji funkcji `do_search`, które zostały przedstawione na końcu poprzedniej strony, napisali niezbędny kod. Zrobili to zgodnie z prawdziwym stylem serii *Rusz głową!*, korzystając z magnesików do kodowania i... lodówki (lepiej nie pytaj o szczegóły). Impreza, którą postanowili uczcić swój niebywały sukces, trochę wymknęła się spod kontroli i jeden z *redaktorów serii* wpadł na lodówkę (śpiewając *piosenkę o piwie*), w wyniku czego magnesiki rozsypały się po podłodze. Twoim zadaniem jest przymocowanie ich z powrotem w odpowiednich miejscach w kodzie.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

@app.route('/search4', methods=['POST'])
def do_search() ->   ........................ :
    phrase = request.form['phrase']
    letters = request.form['letters']

     ..............................................................
     ..............................................................
     return  ..............................................................
                                  ..............................
                                  ..............................
                                  ..............................
                                  ..............................

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Witamy na stronie internetowej search4letters!')

app.run(debug=True)
```

Zdecyduj, w których miejscach wskazywanych kropkowanymi liniami mają się znaleźć poszczególne magnesiki z kodem.

Tutaj znajdują się magnesiki, z których powinieneś skorzystać.

- `str(search4letters(phrase, letters))`
- `the_letters=letters,`
- `'html'`
- `=`
- `title`
- `=`
- `the_results=results,`
- `)`
- `results`
- `the_phrase=phrase,`
- `the_title=title,`
- `'Oto Twoje wyniki:'`
- `render_template('results.html',`

Magnesiki szablonów. Rozwiązanie

Odnotowawszy skwapliwie, aby na przyszłość mieć baczne oko na konsumpcję piwa przez niektórych redaktorów serii, zabrałeś się do roboty polegającej na odtworzeniu układu wszystkich rozsypanych magnesików z kodem tworzącym zaktualizowaną funkcję `do_search`. Twoim zadaniem było z powrotem przymocować te magnesiki we właściwych miejscach w kodzie programu.

Poniżej możesz się przekonać, do czego udało nam się dojść, gdy wykonaliśmy to zadanie.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html'
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_phrase=phrase,
                           the_letters=letters,
                           the_title=title,
                           the_results=results, )

@app.route('/entry')
    return render_template('entry.html',
                           the_title='Witamy na stronie internetowej search4letters!')

app.run(debug=True)
```

Zmień adnotację, aby wskazać, że ta funkcja zwraca teraz kod HTML, a nie zwykły łańcuch znakowy (jak miało to miejsce w poprzedniej wersji kodu).

Utwórz zmienną Pythona o nazwie „title"...

...i przypisz jej odpowiedni łańcuch znakowy.

Utwórz kolejną zmienną Pythona — o nazwie „results"...

...i przypisz jej wynik wywołania funkcji search4letters przetworzony na łańcuch znakowy.

Wyrenderuj szablon results.html. Pamiętaj, że ten szablon oczekuje czterech wartości argumentów.

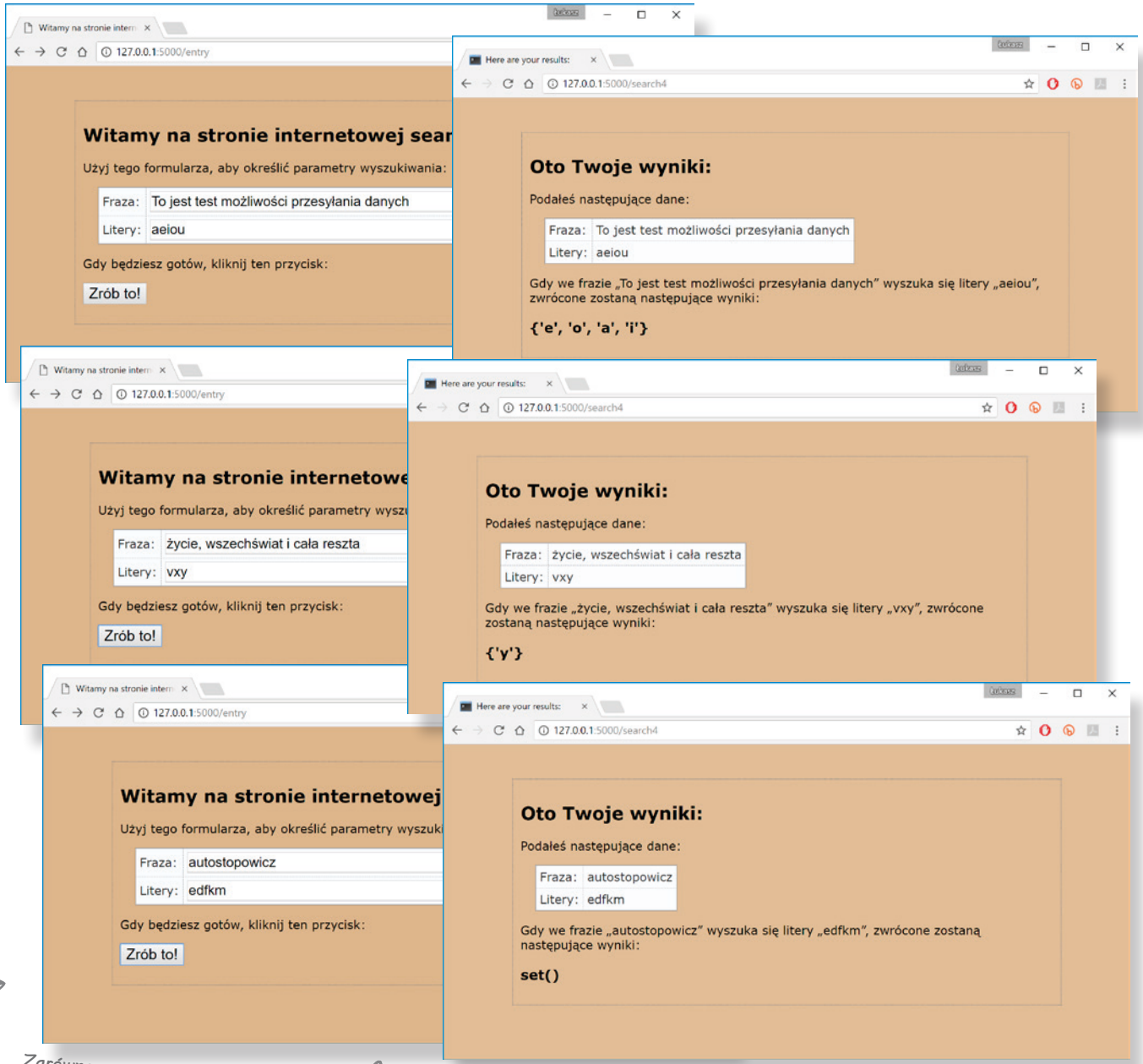
Każda zmienna Pythona jest przypisywana odpowiadającemu jej argumentowi Jinja2. Dzięki temu dane pochodzące z kodu naszego programu są przekazywane do szablonu.

Nie zapomnij o nawiasie zamykającym wywołanie funkcji.

Teraz, gdy magnesiki trafiły już na odpowiednie miejsca, wprowadź przedstawione powyżej zmiany w swojej kopii kodu zapisanej w pliku *vsearch4web.py*. Pamiętaj, aby zapisać swój plik, co spowoduje automatyczne przeładowanie Twojej aplikacji WWW przez framework Flask. W tym momencie jesteśmy gotowi na przeprowadzenie kolejnego testu.

Jazda próbna

Przetestujmy nową wersję naszej aplikacji WWW, używając tych samych przykładów, z których korzystaliśmy we wcześniejszej części tego rozdziału. Zauważ, że framework Flask zrestartował Twoją aplikację w chwili, gdy zapisałeś swój kod.



Zarówno nasza strona wprowadzania danych, jak i strona prezentująca wyniki wyglądają teraz dobrze.

Dodawanie ostatnich szlifów

Raz jeszcze przyjrzyjmy się kodowi, który tworzy teraz program *vsearch4web.py.* Mamy nadzieję, że rozumiesz już należycie cały ten kod. Jeden niewielki element syntaktyczny często budzi wątpliwości programistów przechodzących do Pythona z innych języków programowania. Tym elementem jest włączenie w wywołaniu funkcji `render_template` końcowego znaku przecinka, ponieważ programiści ci mają wrażenie, że powinien on spowodować wystąpienie błędu składni i zwyczajnie nie powinien się tam znajdować. Mimo że rzeczywiście wygląda to trochę dziwnie (na pierwszy rzut oka), język Python dopuszcza użycie tego przecinka, ale wcale go nie wymaga, dlatego możemy bezpiecznie przejść dalej i nie martwić się nim więcej.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

app.run(debug=True)
```

Ten dodatkowy przecinek wygląda trochę dziwnie, ale jest absolutnie poprawnym (choć opcjonalnym) elementem składni języka Python.

W tej wersji naszej aplikacji WWW obsługujemy następujące ciągi URL: /, `/search4` oraz `/entry`; część z nich sięga swoją historią pierwszej wersji aplikacji wykorzystującej framework Flask, którą utworzyliśmy (zaraz na początku niniejszego rozdziału). W tej chwili wywołanie URL-a / powoduje wyświetlenie całkiem przyjaznego, lecz niewiele wnoszącego komunikatu „Witaj, świecie, tu Flask!”.

Moglibyśmy usunąć ten ciąg URL z kodu wraz z powiązaną z nim funkcją `hello` (jako że nie potrzebujemy już ani jednego, ani drugiego), lecz gdy to zrobimy, w oknie dowolnej przeglądarki łączącej się z naszą aplikacją przy użyciu ciągu /, który jest domyślnym URL-em w przypadku większość aplikacji i serwisów WWW, pojawi się komunikat błędu 404 „Not found” (nie znaleziony). Aby uniknąć tego denerwującego komunikatu błędu, poprośmy framework Flask o przekierowanie wszelkich żądań odwołujących się do ciągu / do URL-a `/entry`. Można to zrobić, zmieniając funkcję `hello` w taki sposób, aby za pomocą funkcji `redirect` zwracała przekierowanie HTML do każdej przeglądarki internetowej, która zażąda URL-a /, co w praktyce będzie oznaczało każdorazowe podstawienie ciągu `/entry` pod dowolne żądanie dotyczące URL-a /.

Przekierowanie pozwalające uniknąć niechcianych błędów

Aby skorzystać z możliwości przekierowania oferowanych przez framework Flask, do wiersza zaczynającego się od konstrukcji `from flask`, który znajduje się na samym początku Twojego kodu, dodaj instrukcję importowania funkcji `redirect`, a następnie zmień kod funkcji `hello` w taki sposób, aby wyglądał jak przedstawiony poniżej.

Dodaj funkcję redirect do listy importowanych elementów.

```
from flask import Flask, render_template, request, redirect
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> '302':
   return redirect('/entry')
      ...
```

Zmień adnotację w taki sposób, aby jasno wskazywała, co będzie zwracane przez tę funkcję. Przypomnij sobie, że kody stanu HTTP z przedziału od 300 do 399 oznaczają przekierowania, a kod 302 to właśnie ten, który framework Flask odsyła przeglądarce, gdy wywoływana jest funkcja redirect.

Reszta kodu pozostaje bez zmian.

Wywołaj zapewnianą przez framework Flask funkcję redirect, aby zażądać alternatywnego URL-a (w tym przypadku będzie to ciąg /entry).

Dzięki tej niewielkiej zmianie w kodzie użytkownikom naszej aplikacji WWW pokazywany jest formularz HTML niezależnie od tego, czy zażądali oni URL-a `/entry`, czy też `/`.

Dokonaj tej modyfikacji, zapisz swój kod (co spowoduje automatyczne przeładowanie aplikacji), a następnie spróbuj przejść pod obydwa URL-e, korzystając ze swojej przeglądarki internetowej. Za każdym razem powinieneś zobaczyć formularz HTML. Przyjrzyj się komunikatom stanu wyświetlanym przez aplikację w wierszu poleceń. Powinieneś zobaczyć tam coś podobnego do tego, co pokazano poniżej.

Żądanie dotyczy URL-a /entry i zostaje natychmiast obsłużone. Zauważ widoczny tu kod stanu 200 (i przypomnij sobie z lektury wcześniejszej części tego rozdziału, że kody z przedziału od 200 do 299 to komunikaty powodzenia, oznaczające, że serwer przyjął, zrozumiał i przetworzył żądanie klienta).

```
      ...
* Detected change in 'C:\\webapp\\vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 103-121-075
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [16/Jun/2017 08:46:39] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [16/Jun/2017 08:46:44] "GET / HTTP/1.1" 302 -
127.0.0.1 - - [16/Jun/2017 08:46:44] "GET /entry HTTP/1.1" 200 -
```

Zapisałeś swój kod, więc Flask przeładował Twoją aplikację WWW.

Jeśli żądanie dotyczy URL-a /, nasza aplikacja WWW najpierw odpowiada kodem 302 oznaczającym przekierowanie, a następnie przeglądarka wysyła kolejne żądanie o URL /entry, które zostaje już z powodzeniem obsłużone przez aplikację (zwróć uwagę na drugi kod stanu 200).

Jak widać, nasze przekierowanie świetnie spełnia swoją rolę, ale jest trochę nieekonomiczne, ponieważ pojedyncze żądanie dotyczące URL-a / za każdym razem zmienia się w dwa żądania (mimo że może nam tu trochę pomóc pamięć podręczna działająca po stronie klienta, rozwiązanie to wciąż nie jest optymalne). Byłoby cudownie, gdyby tylko framework Flask pozwalał w jakiś sposób połączyć więcej niż jeden URL z określoną funkcją, całkowicie zapobiegając potrzebie stosowania przekierowania, nie sądzisz?

Z funkcjami może być powiązanych wiele URL-i

Nietrudno zgadnąć, dokąd tu zmierzamy, prawda?

Okazuje się, że Flask rzeczywiście może powiązać z daną funkcją więcej niż jeden URL, co pozwala ograniczyć konieczność stosowania przekierowań podobnych do tego, które zostało przedstawione na poprzedniej stronie. Jeśli z funkcją powiązanych jest więcej URL-i, framework próbuje dopasować każdy z nich po kolei, a gdy mu się to uda, funkcja jest uruchamiana.

Nietrudno jest skorzystać z tej możliwości. Na początek usuń funkcję `redirect` z listy elementów importowanych w wierszu zaczynającym się od konstrukcji `from flask` i umieszczonym na samej górze kodu Twojego programu. Nie będziemy jej już potrzebować, więc lepiej nie importować czegoś, czego i tak nie mamy zamiaru używać. Następnie, korzystając ze swojego edytora, wytnij wiersz kodu `@app.route('/')` i wklej go powyżej wiersza `@app.route('/entry')`, który znajduje się w okolicach końca pliku. Na zakończenie usuń dwa wiersze kodu tworzącego funkcję `hello`, ponieważ nie są już potrzebne w Twojej aplikacji.

Gdy zakończysz wprowadzanie tych zmian, kod Twojego programu powinien wyglądać tak, jak poniżej.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

app.run(debug=True)
```

Nie musimy już importować funkcji redirect, więc usunęliśmy ją z wiersza importowania.

Funkcja hello została usunięta.

Z funkcją związaną ze stroną wprowadzania danych powiązane są teraz dwa ciągi URL.

Zapisanie tego kodu (powodujące przeładowanie aplikacji) umożliwia nam przetestowanie tej nowej funkcji programu. Gdy odwiedzisz URL /, w oknie przeglądarki ukaże się formularz HTML. Szybkie spojrzenie na komunikaty stanu generowane przez Twoją aplikację WWW potwierdzą, że przetwarzanie URL-a / wymaga teraz już tylko jednego żądania, w przeciwieństwie do dwóch, jak miało to miejsce wcześniej.

```
...
* Detected change in 'C:\\webapp\\vsearch4web.py', reloading
  * Restarting with stat
  * Debugger is active!
* Debugger PIN: 103-121-075
  * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [16/Jun/2017 09:29:36] "GET /entry HTTP/1.1" 200 -
```

Zapisanie nowej wersji kodu jak zawsze powoduje przeładowanie naszej aplikacji WWW.

Jedno żądanie, jedna odpowiedź. O to nam chodziło :)

Odświeżanie tego, co wiemy

Ostatnich 40 stron poświęciliśmy na tworzenie niewielkiej aplikacji WWW, która udostępnia możliwości oferowane przez funkcję `search4letters` w sieci (za pośrednictwem prostej strony internetowej). W tej chwili aplikacja ta działa lokalnie na Twoim komputerze. Niebawem omówimy sposób umieszczania jej w chmurze, teraz jednak przypomnijmy, czego dowiedziałeś się do tej pory.

CELNE SPOSTRZEŻENIA

- Dowiedziałeś się o indeksie pakietów Pythona (**PyPI**), stanowiącym scentralizowane repozytorium niezależnych modułów Pythona. Pakiety pochodzące z PyPI możesz automatycznie instalować za pomocą narzędzia `pip`, gdy Twój komputer podłączony jest do internetu.
- Korzystałeś z narzędzia `pip`, aby zainstalować niewielki framework sieciowy o nazwie **Flask**, którego używałeś następnie do zbudowania własnej aplikacji WWW
- Wartość `__name__` (utrzymywana przez interpreter) określa przestrzeń nazw aktywną w danej chwili (więcej na ten temat już wkrótce).
- Symbol `@` umieszczony przed nazwą funkcji wskazuje, że jest ona **dekoratorem**. Dekoratory umożliwiają zmianę działania istniejących funkcji bez konieczności modyfikowania ich kodu. W swojej aplikacji WWW korzystałeś z zapewnianego przez framework Flask dekoratora `@app.route`, aby powiązać ciągi URL z funkcjami Pythona. Funkcję można dekorować więcej niż jeden raz (o czym miałeś okazję się przekonać na przykładzie funkcji `do_search`).
- Nauczyłeś się używać silnika szablonów tekstowych **Jinja2** do renderowania stron HTML z wnętrza swojej aplikacji WWW.

Czy to już wszystko w tym rozdziale?

Wybaczymy Ci, jeśli sądzisz, że rozdział ten nie wniósł wiele nowego do Twojej wiedzy na temat języka Python. Rzeczywiście nie wniósł. Jednak jednym z celów, jakie przyświecały nam przy pisaniu niniejszego rozdziału, było pokazanie Ci, jak zaledwie kilka wierszy kodu Pythona umożliwia utworzenie czegoś w ogólny sposób użytecznego w sieci, w czym niemała zasługa frameworka Flask. Bardzo pomogło nam w tym również zastosowanie technologii szablonów, ponieważ pozwoliło to odseparować kod Pythona (tworzący logikę aplikacji WWW) od stron HTML (tworzących interfejs użytkownika aplikacji).

Zwiększenie możliwości naszej aplikacji WWW nie wymaga tytanicznej pracy. W gruncie rzeczy jakieś cudowne dziecko HTML-a mogłoby opracować dla Ciebie więcej stron, podczas gdy Ty skupiłbyś się na pisaniu kodu Pythona, za pomocą którego wszystko zostałoby połączone w jedną całość. Wraz z rozrostem Twojej aplikacji WWW to rozdzielenie obowiązków naprawdę zaczyna się opłacać. Ty możesz koncentrować się na kodzie Pythona (odgrywając rolę programisty w projekcie), podczas gdy HTML-owy magik skupia się na tworzeniu kodu HTML (ponieważ jest to jego działka). Oczywiście każdy z Was musi dowiedzieć się nieco na temat szablonów silnika Jinja2, ale chyba nie jest to szczególnie trudne, prawda?

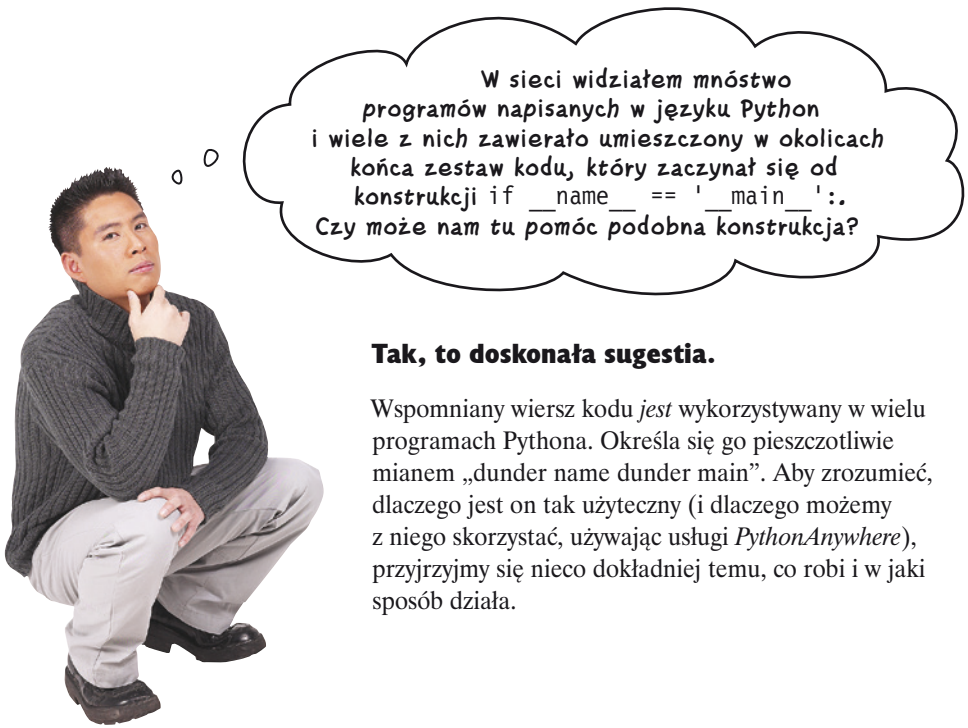
Przygotowywanie aplikacji WWW do umieszczenia w chmurze

Skoro Twoja aplikacja WWW działa już zgodnie ze specyfikacją, lecz jak dotąd tylko lokalnie na Twoim komputerze, pora pomyśleć o umieszczeniu jej w miejscu, w którym będzie dostępna dla szerszego grona odbiorców. Mamy tu do wyboru mnóstwo opcji, ponieważ programiści Pythona mogą korzystać z wielu możliwości hostingu swoich rozwiązań. Bardzo popularna jest wykorzystująca technologię chmury usługa, która jest utrzymywana w ramach AWS i nosi nazwę *PythonAnywhere*. Kochamy ją w naszym laboratorium *Rusz głową!*.

Podobnie jak ma to miejsce w przypadku niemal wszystkich chmurowych rozwiązań tego rodzaju, *PythonAnywhere* chce mieć kontrolę nad sposobem uruchamiania Twojej aplikacji. Dla Ciebie oznacza to tyle, że usługa ta bierze odpowiedzialność za wywołanie metody `app.run()` w Twoim imieniu, co przekłada się na to, że nie musisz już wywoływać jej samodzielnie w swoim kodzie. W rzeczywistości, gdy spróbujesz wykonać ten wiersz kodu, usługa *PythonAnywhere* po prostu odmówi uruchomienia Twojej aplikacji WWW.

Prostym rozwiązaniem tego problemu byłoby usunięcie ostatniego wiersza kodu z Twojego pliku *przed* wdrożeniem go w chmurze. Rozwiązanie to z pewnością by zadziałało, ale oznaczałoby też, że musiałbyś zawsze dodawać ten wiersz kodu z powrotem, gdy tylko zaszłaby konieczność lokalnego uruchomienia aplikacji. Gdy piszesz i testujesz nowy kod, zawsze powinieneś robić to lokalnie (a nie w ramach serwisu *PythonAnywhere*), ponieważ chmury powinieneś używać wyłącznie do wdrażania aplikacji, nie do jej tworzenia. Pozbycie się wiersza kodu oznaczałoby ponadto, że będziesz musiał utrzymywać dwie wersje tej samej aplikacji WWW: jedną zawierającą ten wiersz i jedną bez niego. Nigdy nie jest to dobrym pomysłem (i staje się coraz trudniejsze w praktyce, gdy zaczynają się mnożyć zmiany).

Byłoby miło, gdyby istniał jakiś sposób wybiórczego wykonywania bazy kodu w zależności od tego, czy uruchamiasz swoją aplikację WWW lokalnie na swoim komputerze, czy też zdalnie za pomocą usługi *PythonAnywhere*...



Tak, to doskonała sugestia.

Wspomniany wiersz kodu *jest* wykorzystywany w wielu programach Pythona. Określa się go pieszczotliwie mianem „dunder name dunder main”. Aby zrozumieć, dlaczego jest on tak użyteczny (i dlaczego możemy z niego skorzystać, używając usługi *PythonAnywhere*), przyjrzyjmy się nieco dokładniej temu, co robi i w jaki sposób działa.

Dunder name dunder main z bliska

Aby zrozumieć konstrukcję programistyczną przedstawioną na końcu poprzedniej strony, przyjrzyjmy się niewielkiemu programowi o nazwie *dunder.py*, w którym jest ona wykorzystywana. Ten składający się z trzech wierszy program zaczyna pracę od wyświetlenia na ekranie komunikatu zawierającego nazwę aktywnej w danej chwili przestrzeni nazw, która jest przechowywana w zmiennej `__name__`. Za pomocą instrukcji `if` sprawdza następnie, czy wartością zmiennej `__name__` jest `__main__`, a gdy tak jest, wyświetla kolejny komunikat potwierdzający wartość zmiennej `__name__` (to znaczy, wykonuje zestaw kodu związany z instrukcją `if`).

Program dunder.py; wszystkie trzy tworzące go wiersze.

```
print('Zaczynamy od:', __name__)
if __name__ == '__main__':
    print('I kończymy z:', __name__)
```

Wyświetla wartość zmiennej __name__.

Wyświetla wartość zmiennej __name__, jeśli jest to __main__.

Skorzystaj ze swojego ulubionego edytora tekstowego (lub środowiska IDLE), aby utworzyć plik *dunder.py*, a następnie uruchom ten program w wierszu poleceń i przekonaj się, co się stanie. Jeśli korzystasz z systemu *Windows*, użyj polecenia pokazanego poniżej.

```
C:\> py  -3  dunder.py
```

Jeśli korzystasz z systemu *Linux* lub *Mac OS X*, użyj poniższego polecenia.

```
$ python3  dunder.py
```

Niezależnie od tego, jakiego systemu operacyjnego używasz, gdy program *dunder.py* uruchomisz *bezpośrednio* za pomocą Pythona, na ekranie pojawią się komunikaty przedstawione poniżej.

```
Zaczynamy od: __main__
I kończymy z: __main__
```

Gdy program wykonywany jest bezpośrednio przez Pythona, obydwa wywołania funkcji print wyświetlają dane wyjściowe.

Jak dotąd świetnie.

Sprawdź teraz, co się stanie, gdy zaimportujemy plik *dunder.py* (który — jak zapewne doskonale pamiętasz — jest równocześnie modułem) w wierszu poleceń powłoki Pythona. Poniżej prezentujemy Ci, co się stanie w systemie *Windows*. Jeśli chcesz zobaczyć, jak sprawy mają się w systemie *Linux* lub *Mac OS X*, zastąp polecenie `py -3` poleceniem `python3`.

```
C:\Users\lsuma\Desktop\Python\R05>py -3
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import dunder
Zaczynamy od: dunder
```

Patrz na to: mamy tu wyświetlony tylko jeden wiersz (w przeciwieństwie do dwóch, jak miało to miejsce wcześniej), ponieważ zmienna __name__ otrzymała wartość dunder (będącą nazwą zaimportowanego modułu).

Co powinieneś stąd zapamiętać: gdy kod Twojego programu jest uruchamiany *bezpośrednio* przez Pythona, instrukcja `if`, podobna do tej, która znajduje się w programie *dunder.py*, zwróci wartość `True`, ponieważ aktywną przestrzenią nazw jest `__main__`. Jeśli jednak kod programu zostanie zaimportowany jako moduł (jak ma to miejsce w przedstawionym powyżej przykładzie z wierszem powłoki Pythona), instrukcja `if` zawsze zwróci wartość `False`, ponieważ wartością zmiennej `__name__` nie jest `__main__`, lecz nazwa zaimportowanego modułu (w tym przypadku jest to `dunder`).

Wykorzystywanie dunder name dunder main w praktyce

Teraz, gdy wiesz już, co umożliwia konstrukcja *dunder name dunder main*, skorzystajmy z niej, aby rozwiązać problem wiążący się z tym, że usługa *PythonAnywhere* chce wykonywać metodę `app.run()` w naszym imieniu.

Okazuje się, że usługa *PythonAnywhere* uruchamia kod naszej aplikacji WWW, importując plik zawierający jej kod i traktując go jak każdy inny moduł. Jeśli operacja importu się powiedzie, usługa *PythonAnywhere* wywoła metodę `app.run()`. Tłumaczy to, dlaczego pozostawienie tego wywołania na końcu naszego kodu stanowi problem dla tej usługi. Zakłada ona bowiem, że wywołanie metody `app.run()` nie miało jeszcze miejsca, i nie udaje jej się uruchomić naszej aplikacji WWW, gdy wywołanie to już się odbyło.

Aby obejść ten problem, opakuj wywołanie metody `app.run()` w instrukcję `if` zawierającą konstrukcję *dunder name dunder main* (zapewniając w ten sposób, że metoda ta nie wykona się nigdy wówczas, kiedy kod aplikacji WWW zostanie zaimportowany).

Przystąp do edycji pliku *vsearch4web.py* po raz ostatni (w każdym razie w tym rozdziale) i zastąp ostatni wiersz jego kodu przedstawionym fragmentem.

```
if __name__ == '__main__':
    app.run(debug=True)
```

Wiersz kodu zawierający wywołanie metody app.run() będzie wykonany tylko wtedy, gdy program zostanie uruchomiony bezpośrednio przez Pythona.

Ta niewielka modyfikacja umożliwi Ci bezproblemowe uruchamianie aplikacji WWW lokalnie (w tym przypadku wiersz zawierający wywołanie metody `app.run()` *będzie* wykonywany), jak również wdrożenie jej w ramach usługi *PythonAnywhere* (w tym przypadku wiersz zawierający wywołanie metody `app.run()` *nie będzie* wykonywany). Niezależnie od tego, gdzie Twoja aplikacja WWW będzie uruchamiana, masz teraz tylko jedną wersję kodu, która doskonale radzi sobie w każdych warunkach.

Wdrażanie aplikacji WWW przy użyciu usługi PythonAnywhere (cóż, prawie)

W tym momencie musisz już tylko przeprowadzić właściwe wdrożenie w chmurowym środowisku *PythonAnywhere*.

Pamiętaj, że dla celów związanych z tą książką wdrożenie Twojej aplikacji WWW w środowisku chmurowym *nie* jest bezwzględnie wymagane. Pomimo że w kolejnym rozdziale zamierzamy rozszerzyć program *vsearch4web.py* o dodatkowe możliwości, w gruncie rzeczy nie musisz wdrażać aplikacji za pomocą usługi *PythonAnywhere*, aby kontynuować lekturę i modyfikowanie kodu. Nadal możesz spokojnie edytować, uruchamiać i testować swoją aplikację WWW lokalnie i nie będziesz miał z tego tytułu żadnych problemów w dalszej części książki.

Jeśli jednak naprawdę chcesz wdrożyć swoją aplikację w chmurze, przejdź do dodatku B, w którym znajdziesz przedstawione krok po kroku instrukcje dotyczącego tego, jak przeprowadzić wdrożenie przy użyciu usługi *PythonAnywhere*. Nie jest to zbyt trudne zadanie i nie powinno Ci zająć więcej niż 10 minut.

Niezależnie od tego, czy zdecydujesz się na wdrożenie w chmurze, czy też nie, widzimy się w następnym rozdziale, w którym zaczniemy przyglądać się niektórym możliwościom pozwalającym zapisywać dane pochodzące z Twoich programów opracowanych w języku Python.

Kod przykładów z rozdziału 5.

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
   return 'Witaj, świecie, tu Flask!'

@app.route('/search4')
def do_search() -> str:
   return str(search4letters('życie, wszechświat i cała reszta', 'eiru,!'))

app.run()
```

To jest kod programu hello_flask.py, naszej pierwszej aplikacji WWW wykorzystującej framework Flask (jeden z dostępnych w Pythonie mikroframeworków sieciowych).

To jest kod programu search4web.py. Ta aplikacja WWW udostępniała w sieci możliwości zapewniane przez funkcję search4letters. Oprócz frameworka Flask wykorzystaliśmy tutaj silnik szablonów Jinja2.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej
search4letters!')

if __name__ == '__main__':
   app.run(debug=True)
```

To jest kod programu dunder.py, który pomógł nam zrozumieć bardzo wygodny mechanizm „dunder name dunder main".

```
print('Zaczynamy od:', __name__)
if __name__ == '__main__':
   print('I kończymy z:', __name__)
```

Gdzie umieścić Twoje dane

Wcześniej czy później będziesz musiał gdzieś bezpiecznie zachować swoje dane.

A gdy idzie o **przechowywanie danych**, język Python daje Ci wszystko, czego trzeba. W tym rozdziale poznasz sposoby zapisywania danych w *plikach tekstowych* i odczytywania ich z tych plików. Pliki tekstowe mogą się wydawać dość prostym rozwiązaniem, gdy weźmie się pod uwagę rozwój mechanizmów do przechowywania danych, nadal są jednak z powodzeniem stosowane w wielu różnych sytuacjach. Oprócz zapisywania danych w plikach i ich odczytywania przybliżymy Ci również pewne sztuczki związane z manipulowaniem danymi. Z prezentacją „poważnego materiału" (czyli informacji na temat przechowywania danych w bazie danych) poczekamy do następnego rozdziału, ale do tej pory mamy jeszcze do omówienia tak dużo tematów związanych z korzystaniem z plików, że na pewno nie będziemy się nudzić.

Robienie czegoś z danymi Twojej aplikacji WWW

Obecnie Twoja aplikacja WWW (opracowana w rozdziale 5.) przyjmuje dane wejściowe (w postaci frazy i liter) z dowolnej przeglądarki internetowej, wywołuje metodę `search4letters`, a następnie zwraca uzyskane od niej wyniki oczekującej przeglądarce. Po wykonaniu swojego zadania aplikacja pozbywa się wszystkich danych, które przetwarzała.

Istnieje sporo pytań dotyczących danych używanych przez naszą aplikację WWW, które moglibyśmy zadać. Na przykład: *Na ile żądań udało jej się odpowiedzieć? Jaka jest najczęściej używana lista liter? Z których adresów IP nadchodzą żądania? Z której przeglądarki internetowej najczęściej korzystają użytkownicy naszej aplikacji?* I tak dalej, i tym podobne.

Aby zacząć odpowiadać na te (i inne) pytania, powinniśmy zapisywać dane naszej aplikacji WWW, zamiast po prostu się ich pozbywać. Powyższa sugestia ma sens: logujmy dane dotyczące każdego żądania, a następnie — gdy mechanizm logowania będzie już gotowy — zabierzmy się do udzielania odpowiedzi na pytania, które zadaliśmy.

Python obsługuje otwieranie, przetwarzanie i zamykanie

Niezależnie od używanego języka programowania najprostszym sposobem umożliwiającym przechowywanie danych jest zapisanie ich w pliku tekstowym. Dlatego też Python oferuje wbudowane możliwości obsługi operacji *otwierania*, *przetwarzania* i *zamykania* (ang. *open*, *process*, *close*). Dzięki temu możesz **otworzyć** plik, **przetworzyć** przechowywane w nim dane w jakiś sposób (odczytać, zapisać lub dodać dane), a następnie **zamknąć** plik, gdy zakończysz wymagane działania (co spowoduje zapisanie Twoich zmian).

Poniżej znajdziesz opis sposobu, w jaki możesz korzystać z oferowanego przez Pythona mechanizmu *otwierania*, *przetwarzania* i *zamykania*, aby otworzyć plik, przetworzyć go, dodając do niego krótkie łańcuchy znakowe, a następnie go zamknąć. Ponieważ zamierzamy tu tylko eksperymentować, wykonajmy nasz kod w wierszu poleceń powłoki Pythona.

Zaczniemy od wywołania funkcji `open` dla pliku o nazwie *todos.txt* przy użyciu *trybu dołączania* (ang. *append mode*), ponieważ planujemy dodać dane do tego pliku. Jeśli to wywołanie się powiedzie, interpreter zwróci nam obiekt (znany również jako *strumień pliku*; ang. *file stream*), będący synonimem prawdziwego pliku. Obiekt ten zostanie przypisany do zmiennej o nazwie `todos` (możesz ją zmienić).

Otwórz plik...

...o tej nazwie...

```
>>> todos = open('todos.txt', 'a')
```

Jeśli wszystko jest w porządku, funkcja open zwraca strumień pliku, który przypisaliśmy do tej zmiennej.

...w trybie dołączania.

Porady maniaka

Aby uzyskać dostęp do wiersza poleceń Pythona (>>>), wykonaj jedną z poniższych czynności:

- uruchom środowisko IDLE na swoim komputerze;
- wykonaj polecenie `python3` w terminalu systemu operacyjnego *Linux* lub *Mac OS X*;
- skorzystaj z polecenia `py -3` w wierszu poleceń systemu *Windows*.

Zmienna `todos` umożliwia Ci odwoływanie się do Twojego pliku w kodzie (w innych językach programowania używa się tu określenia *uchwyt pliku*; ang. *file handle*). Gdy nasz plik jest już otwarty, dokonajmy w nim zapisu, korzystając z funkcji `print`. Zauważ teraz, że przedstawione powyżej wywołanie tej funkcji zawiera dodatkowy argument (o nazwie `file`) wskazujący strumień pliku, w którym realizowany ma być zapis. Musimy zapamiętać trzy rzeczy do zrobienia (choć tak naprawdę lista rzeczy do zrobienia nigdy się nie kończy), dlatego wywołujemy funkcję `print` trzykrotnie.

Zapisujemy komunikat...

...do strumienia pliku[1].

```
>>> print('Pozmywaj naczynia.', file=todos)
>>> print('Nakarm kota.', file=todos)
>>> print('Przygotuj zeznanie podatkowe.', file=todos)
```

Ponieważ nie mamy już nic do dodania do naszej listy rzeczy do zrobienia, zamknijmy plik, wywołując na rzecz obiektu strumienia metodę `close`, która jest udostępniana przez interpreter każdemu obiektowi tego rodzaju.

```
>>> todos.close()
```

Zakończyliśmy, więc pora po sobie posprzątać, zamykając strumień pliku.

Jeśli zapomnisz wywołać metodę `close`, *potencjalnie* możesz stracić dane. Pamiętaj o jej wywołaniu.

---

[1] W razie wystąpienia problemów z zapisywaniem łańcuchów znakowych zawierających polskie znaki należy skorzystać z metody `open` oferowanej przez moduł `codecs`. W tym celu trzeba najpierw zaimportować tę metodę, a następnie wywołać ją z dodatkowym argumentem określającym sposób kodowania znaków, tak jak zostało to pokazane poniżej.

```
>>> from codecs import open
>>> todos = open('todos.txt', 'a', 'utf-8')
>>> ...
```

Powinno to wyeliminować problemy związane z kodowaniem polskich znaków diakrytycznych. Dalsze operacje wykonuje się już tak samo, jak pokazano powyżej — *przyp. tłum.*

Odczytywanie danych z istniejącego pliku

Gdy dodałeś już kilka wierszy danych do pliku *todos.txt*, przekonajmy się, jaki kod odpowiedzialny za *otwieranie*, *przetwarzanie* i *zamykanie* jest niezbędny, aby odczytać zapisane dane z naszego pliku i wyświetlić je na ekranie.

Tym razem chcemy jedynie odczytywać dane z pliku, dlatego zamiast otwierać plik w trybie dołączania, powinniśmy skorzystać z trybu odczytu, który jest **domyślnym trybem** działania metody `open`. Oznacza to, że w ogóle nie musimy tu podawać argumentu trybu; funkcji powinniśmy podać jedynie nazwę pliku do otwarcia. W tym kodzie nie będziemy używać zmiennej `todos` w roli synonimu naszego pliku; zamiast tego będziemy się odwoływać do pliku za pomocą zmiennej o nazwie `tasks` (podobnie jak wcześniej, możesz tu użyć dowolnej nazwy, jaka tylko przyjdzie Ci do głowy).

Odczyt to domyślny tryb używany przez funkcję open.

Otwórz plik...

...o tej nazwie.

```
>>> tasks = open('todos.txt')
```

Jeśli wszystko jest w porządku, funkcja open zwraca strumień pliku, który przypisaliśmy do tej zmiennej[2].

Użyjmy teraz zmiennej `tasks` w połączeniu z pętlą `for`, aby odczytać poszczególne wiersze danych z pliku. Gdy to zrobimy, do zmiennej iteracyjnej (o nazwie `chore`) przy każdym przejściu pętli będzie przypisywany jeden (bieżący) wiersz danych odczytany z pliku. Jest tak, ponieważ gdy w Pythonie korzystasz ze strumienia pliku oraz instrukcji pętli `for`, interpreter działa na tyle sprytnie, aby odczytywać po jednym wierszu danych w każdej iteracji tej pętli. Jego spryt przejawia się też w tym, żeby zakończyć działanie pętli, gdy nie ma więcej danych do odczytania.

Traktuj zmienną chore jako synonim wiersza używanego pliku.

```
>>> for chore in tasks:
        print(chore)

Pozmywaj naczynia.

Nakarm kota.

Przygotuj zeznanie podatkowe.
```

Zmienna tasks reprezentuje strumień pliku.

Na ekranie wyświetlane są dane pochodzące z pliku todos.txt. Zauważ, że pętla kończy swoje działanie, gdy w pliku nie ma więcej wierszy do odczytania.

Ponieważ tylko odczytujesz dane z zapisanego już wcześniej pliku, wywołanie metody `close` jest tu o wiele mniej istotne, niż miało to miejsce wtedy, gdy zapisywałeś dane. Jednak zawsze dobrze jest zamykać pliki, jeśli nie są one już potrzebne, dlatego wywołaj metodę `close`, gdy skończysz korzystać z pliku reprezentowanego przez zmienną `todos`.

```
>>> tasks.close()
```

Zakończyliśmy, więc pora po sobie posprzątać, zamykając strumień pliku.

[2] W razie wystąpienia problemów z odczytywaniem z plików łańcuchów znakowych zawierających polskie znaki należy skorzystać ze sposobu przedstawionego w przypisie zamieszczonym na poprzedniej stronie — *przyp. tłum.*

Nie istnieją głupie pytania

P: O co chodzi z tymi dodatkowymi pustymi wierszami na ekranie? W pliku znajdują się trzy wiersze danych, a pętla `for` wyświetliła ich w sumie aż sześć. Co jest grane?

O: Tak, dane wyświetlone przez pętlę `for` wyglądają trochę dziwnie, nieprawdaż? Aby zrozumieć, co się tu dzieje, weź pod uwagę, że funkcja `print` dokłada znak nowego wiersza do wszystkiego, co wyświetla na ekranie; jest to jej *standardowe zachowanie*. Jeśli połączysz to z faktem, że każdy wiersz w pliku kończy się znakiem nowego wiersza (a znak ten jest odczytywany jeszcze jako część składowa bieżącego), okaże się, że w efekcie za każdym razem wyświetlasz dwa znaki nowego wiersza: jeden pochodzący z pliku i jeden dokładany przez funkcję `print`. Aby poinstruować funkcję `print`, że nie ma dołączać znaku nowego wiersza, zmień proste wywołanie `print(chore)` w wywołanie `print(chore, end='')`. Spowoduje to wyłączenie standardowego zachowania funkcji i zaprzestanie wyświetlania przez nią dodatkowych znaków nowego wiersza.

P: Z jakich innych trybów mogę korzystać, posługując się plikami danych?

O: Masz jeszcze kilka do dyspozycji, zebraliśmy je w poniższej ramce *Porady maniaka*. (A tak na marginesie, to świetne pytanie).

Porady maniaka

Pierwszym argumentem funkcji open jest nazwa pliku do przetworzenia. Drugi argument jest **opcjonalny**. Może przyjmować kilka różnych wartości i określa **tryb**, w którym otwierany jest plik. Do trybów należą „odczyt", „zapis" i „dołączanie". Poniżej wymienione zostały najczęściej używane tryby obsługi plików, z których każdy (oprócz `'r'`, czyli odczytu) powoduje utworzenie nowego, pustego pliku, jeśli plik wskazywany przez pierwszy argument funkcji nie istnieje w chwili jej wywołania:

- Wartość `'r'` powoduje otwarcie pliku do **odczytu**. Jest to tryb domyślny, dlatego określenie tej wartości argumentu jest opcjonalne — jeśli nie poda się drugiego argumentu, przyjmowana jest wartość `'r'`. Zakłada się również, że odczytywany plik istnieje w momencie wywołania funkcji.
- Wartość `'w'` powoduje otwarcie pliku do **zapisu**. Jeśli wskazywany plik już istnieje, dane są z niego usuwane przed udostępnieniem pliku użytkownikowi.
- Wartość `'a'` powoduje otwarcie pliku do **dołączania**. Tryb ten umożliwia zachowanie dotychczasowej zawartości pliku i dodawanie nowych danych na jego końcu (porównaj to działanie z tym, jak działa tryb `'w'`).
- Wartość `'x'` powoduje otwarcie **nowego pliku** do zapisu. Jeśli plik wskazywany przez pierwszy argument istnieje w momencie wywołania funkcji, jej działanie kończy się niepowodzeniem (porównaj to zachowanie z działaniem trybów `'w'` oraz `'a'`).

Pliki są domyślnie otwierane w trybie **tekstowym**, w którym zakłada się, że zawierają wiersze danych tekstowych (zakodowanych na przykład z wykorzystaniem standardu ASCII lub UTF-8). Jeśli korzystasz z danych niemających charakteru tekstowego (na przykład z pliku graficznego lub MP3), możesz wybrać tryb **binarny**, dodając literę `'b'` do dowolnego z trybów obsługi pliku (na przykład argument `'wb'` oznacza „zapis w pliku binarnym"). Jeśli użyjesz znaku + w drugim argumencie wywołania funkcji, otworzysz plik do zapisu *i* odczytu (na przykład argument `'w+b'` oznacza „odczyt z pliku binarnego i zapis w nim"). Po więcej informacji na temat funkcji open (w tym także szczegóły związane z innymi argumentami opcjonalnymi) sięgnij do dokumentacji Pythona.

Przejrzałem sporo napisanych w Pythonie projektów, które można znaleźć w serwisie GitHub, i zauważyłem, że w większości z nich przy otwieraniu plików korzysta się z instrukcji with. O co z nią chodzi?

Instrukcja `with` jest wygodniejsza.

Choć korzystanie z funkcji open i metody `close` (oraz jakieś przetwarzanie danych pomiędzy ich wywołaniami) sprawdza się świetnie, większość programistów Pythona unika procedury *otwierania*, *przetwarzania* i *zamykania*, przedkładając nad nią stosowanie instrukcji `with`. Poświęćmy chwilę, aby przekonać się, dlaczego tak jest.

Lepsze otwieranie, przetwarzanie i zamykanie za pomocą instrukcji with

Zanim opiszemy, dlaczego instrukcja `with` jest tak popularna, przyjrzyjmy się kodowi, w którym jest wykorzystywana. Poniżej znajdziesz przedstawiony niedawno (dwie strony wcześniej) kod, który umożliwia odczytywanie bieżącej zawartości pliku *todos.txt* i wyświetlanie jej na ekranie. Zauważ, że zmieniliśmy wywołanie funkcji `print` w taki sposób, aby uniknąć wyświetlania dodatkowych pustych wierszy.

```
tasks = open('todos.txt')
for chore in tasks:
    print(chore, end='')
tasks.close()
```

Otwórz plik, przypisując strumień pliku do zmiennej.

Wykonaj jakieś operacje.

Zamknij plik.

Napiszmy teraz nowy kod, spełniający to samo zadanie, ale przy użyciu instrukcji `with`. Przedstawione poniżej trzy wiersze kodu robią *dokładnie* to samo, co cztery wiersze widoczne powyżej.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

Otwórz plik.

Przypisz strumień pliku do zmiennej.

Wykonaj jakieś operacje (ten fragment kodu jest taki sam jak powyżej).

Zauważyłeś, że czegoś tu brakuje? Nigdzie nie pojawiło się wywołanie metody `close`, ponieważ instrukcja `with` jest wystarczająco „sprytna”, aby pamiętać o wywołaniu metody `close` *w Twoim imieniu*, gdy tylko zestaw kodu się kończy.

Jest to znacznie bardziej przydatne, niż może się początkowo wydawać, ponieważ wielu programistów bardzo często zapomina wywołać metodę `close` po zakończeniu przetwarzania pliku. Nie stanowi to wielkiego problemu, gdy wykonuje się jedynie operacje odczytu, jednak gdy w grę wchodzą operacje zapisu do pliku, pominięcie wywołania metody `close` może potencjalnie powodować *utratę danych* lub ich *uszkodzenie*. Uwalniając Cię od konieczności pamiętania o każdorazowym wywoływaniu metody `close`, instrukcja `with` pozwala Ci skupić się na tym, co naprawdę chcesz zrobić z danymi znajdującymi się w otwartym pliku.

Python obsługuje otwieranie, przetwarzanie i zamykanie. Ale większość programistów korzystających z tego języka woli korzystać z instrukcji with.

Instrukcja with zarządza kontekstem

Instrukcja `with` jest zgodna z wbudowaną w Pythona konwencją kodowania określaną mianem **protokołu zarządzania kontekstem** (ang. *context management protocol*). Dokładny opis tego protokołu pojawi się w dalszej części tej książki. Teraz trzeba Ci tylko wiedzieć, że gdy korzystasz z plików przy użyciu instrukcji `with`, możesz śmiało zapomnieć o wywołaniu metody `close`. Instrukcja ta zarządza kontekstem, w którym uruchamiany jest zestaw kodu, a gdy używasz jej wraz z wywołaniem funkcji `open`, interpreter sprząta po Tobie, wywołując metodę `close` zawsze, kiedy jest to niezbędne.

Zastosujmy w praktyce to, co już wiesz na temat korzystania z plików. Poniżej znajdziesz bieżący kod swojej aplikacji WWW. Przeczytaj go raz jeszcze, zanim dowiesz się, co powinieneś zrobić.

Ćwiczenie

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

if __name__ == '__main__':
   app.run(debug=True)
```

To jest kod programu vsearch4web.py, przedstawionego w rozdziale 5.

Twoim zadaniem jest napisanie nowej funkcji o nazwie `log_request`, która przyjmuje dwa argumenty: `req` oraz `res`. Gdy funkcja ta zostanie wywołana, argumentowi `req` powinien zostać przypisany obiekt bieżącego żądania sieciowego frameworka Flask, a argumentowi `res` — wynik wywołania funkcji `search4letters`. Działanie funkcji `log_request` ma polegać na dołączeniu wartości argumentów `req` i `res` (w postaci jednego wiersza) do pliku o nazwie *vsearch.log*. Pracę zaczęliśmy za Ciebie, tworząc wiersz `def` tej funkcji. Ty powinieneś teraz uzupełnić brakujący kod (wskazówka: użyj instrukcji `with`).

```
def log_request(req: 'flask_request', res: str) -> None:
```

Tutaj napisz zestaw funkcji.

Ćwiczenie Rozwiązanie

Twoim zadaniem było napisanie nowej funkcji o nazwie `log_request`, która przyjmuje dwa argumenty: `req` oraz `res`. Gdy funkcja ta jest wywoływana, argumentowi `req` zostaje przypisany obiekt bieżącego żądania sieciowego frameworka Flask, a argumentowi `res` — wynik wywołania funkcji `search4letters`. Działanie funkcji `log_request` powinno polegać na dołączeniu wartości argumentów `req` i `res` (w postaci jednego wiersza) do pliku o nazwie *vsearch.log*. Pracę zaczęliśmy za Ciebie, a Ty miałeś uzupełnić brakujący kod.

Ta adnotacja może Cię trochę zdziwić. Pamiętaj jednak, że adnotacje funkcji mają być czytane przez innych programistów. Stanowią one dokumentację, a nie wykonywalny kod. Oznacza to, że interpreter Pythona zawsze je ignoruje, dlatego możesz używać tu dowolnych opisów, które uznasz za stosowne.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log :
        print(req, res, file=log)
```

Użyj instrukcji with, aby otworzyć plik vsearch.log w trybie dołączania.

Wywołaj wbudowaną funkcję print, aby zapisać wartości argumentów req oraz res w otwartym pliku.

Zwróć uwagę, że strumień pliku otrzymał w tym kodzie nazwę „log".

W tej adnotacji użyliśmy zapewnianej przez język Python wartości None, aby wskazać, że ta funkcja nie zwraca żadnej wartości.

Wywoływanie funkcji logującej

Teraz, gdy istnieje już funkcja `log_request`, powinniśmy zadać sobie pytanie: kiedy należy ją wywołać?

Cóż, najpierw dodajmy kod funkcji `log_request` do pliku *vsearch4web.py*. Możesz go wstawić w dowolnym miejscu tego piku, ale my postanowiliśmy umieścić go bezpośrednio nad funkcją `do_search` i powiązanym z nim dekoratorem `@app.route`. Zrobiliśmy tak, ponieważ naszą nową funkcję zamierzamy wywoływać z wnętrza funkcji `do_search`, a umieszczenie funkcji wywoływanej bezpośrednio nad wywołującą wydaje się dobrym rozwiązaniem.

Funkcję `log_request` musimy na pewno wywołać przed tym, zanim funkcja `do_search` zakończy swoje działanie, ale już po tym, gdy wyniki zwrócone przez wywołanie funkcji `search4letters` zostaną przypisane do zmiennej `results`. Poniżej znajdziesz fragment kodu funkcji `do_search` z wstawionym w odpowiednim miejscu wywołaniem funkcji `log_request`.

Wywołaj funkcję log_request w tym miejscu.

```
  ...
phrase = request.form['phrase']
letters = request.form['letters']
title = 'Oto Twoje wyniki:'
results = str(search4letters(phrase, letters))
log_request(request, results)
return render_template('results.html',
  ...
```

Szybki przegląd

Zanim weźmiemy w obroty tę najnowszą wersję programu *vsearch4web.py*, sprawdźmy, czy Twój kod wygląda tak samo jak nasz. Oto cała zawartość pliku z wyróżnionymi fragmentami kodu, które zostały ostatnio dodane.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

def log_request(req: 'flask_request', res: str) -> None:
   with open('vsearch.log', 'a') as log:
      print(req, res, file=log)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   log_request(request, results)
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

if __name__ == '__main__':
   app.run(debug=True)
```

Tutaj znajdują się dodane ostatnio fragmenty kodu, których zadaniem jest logowanie każdego żądania sieciowego w pliku o nazwie „vsearch.log".

Być może zauważyłeś, że żadna z funkcji tworzących naszą aplikację WWW nie zawiera komentarzy. To celowy zabieg z naszej strony (ponieważ na stronach tej książki nie ma zbyt wiele miejsca i musieliśmy z czegoś zrezygnować). Zauważ jednak, że wszystkie związane z tą książką pliki kodu, które możesz pobrać ze stron wydawnictwa, zawierają odpowiednie komentarze.

Weź swoją aplikację WWW w obroty

Uruchom tę wersję swojej aplikacji WWW (jeśli to konieczne) w systemowym wierszu poleceń. Jeśli korzystasz z systemu *Windows*, użyj następującego polecenia:

```
C:\webapps> py -3 vsearch4web.py
```

Jeśli używasz systemu *Linux* lub *Mac OS X*, skorzystaj z takiego polecenia:

```
$ python3 vsearch4web.py
```

Gdy aplikacja będzie już działać, zalogujmy jakieś dane, korzystając z formularza HTML.

Jazda próbna

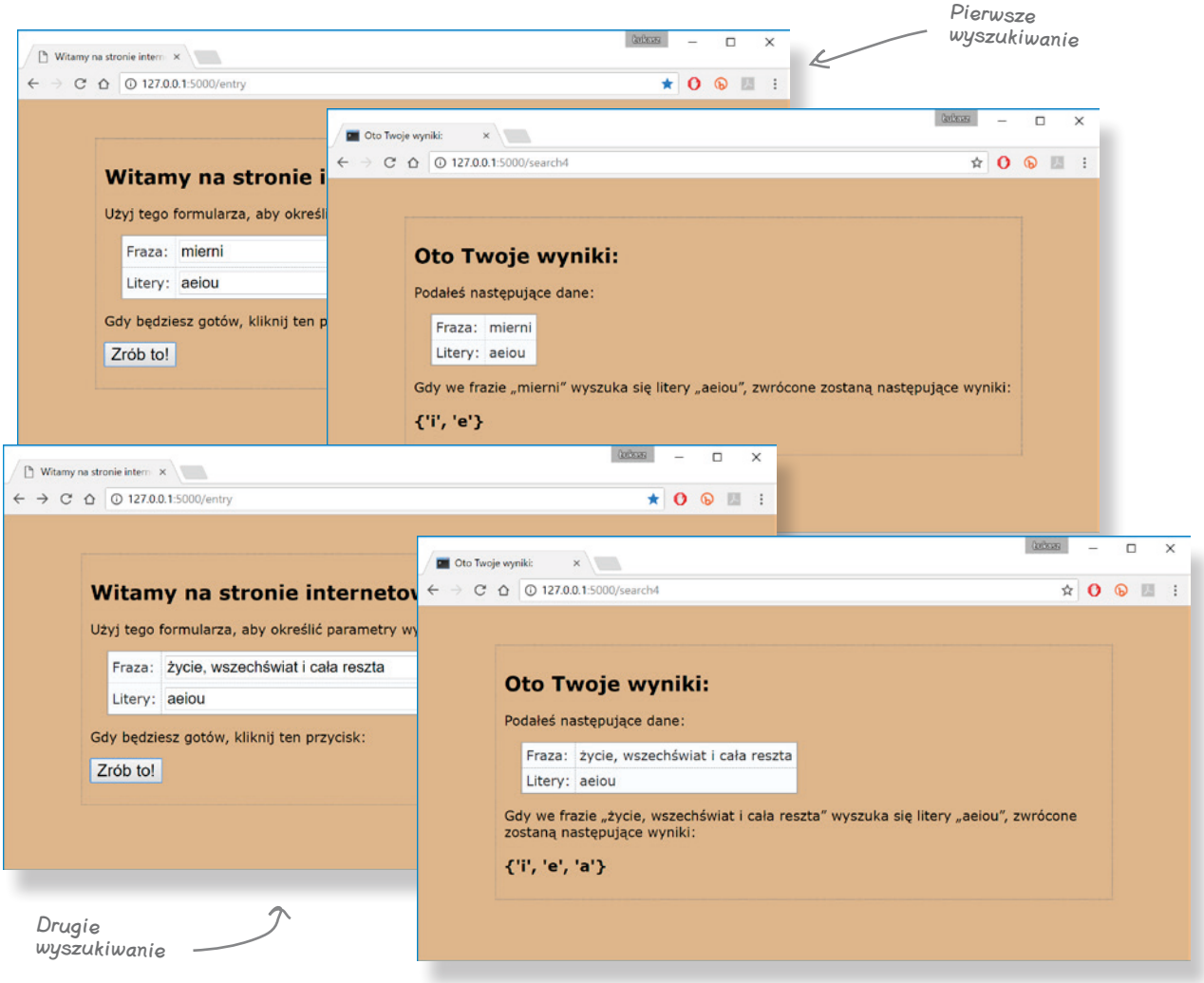
Użyj swojej przeglądarki internetowej, aby wprowadzić jakieś dane w formularzu HTML wykorzystywanym przez aplikację WWW. Jeśli chcesz podążać za tym, co robimy, przeprowadź trzy wyszukiwania, podając kolejno następujące wartości argumentów `phrase` i `letters`:

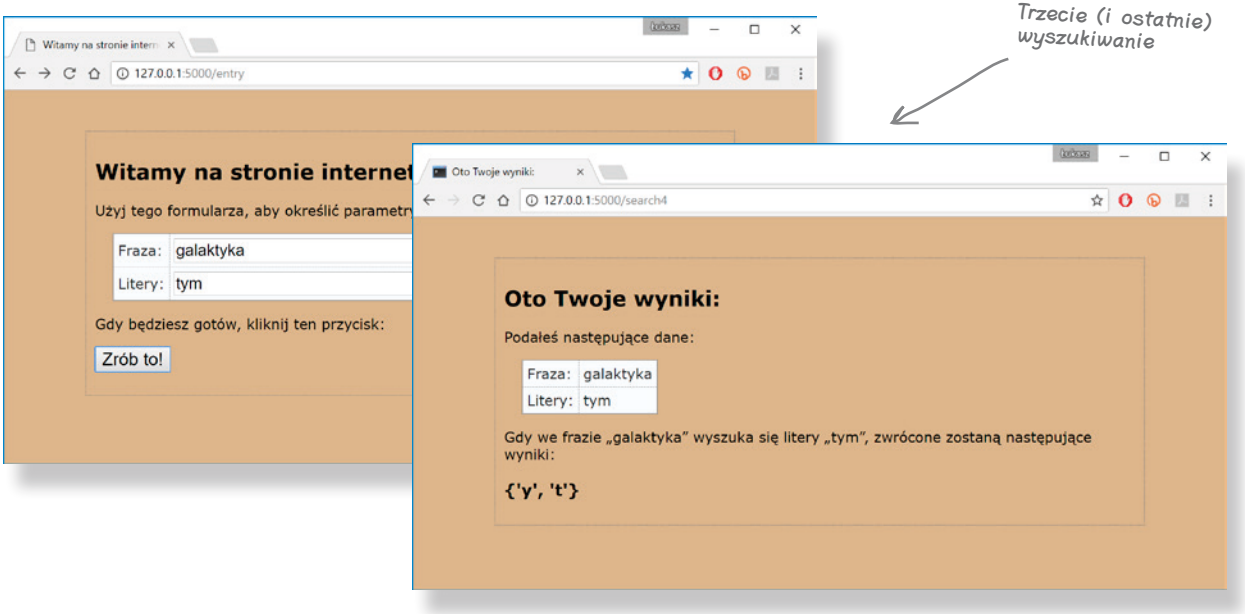
fraza: `mierni`, litery: `aeiou`,

fraza: `życie, wszechświat i cała reszta`, litery: `aeiou`,

fraza: `galaktyka`, litery: `tym`.

Zanim zaczniesz, zauważ, że plik *vsearch.log* na tym etapie jeszcze nie istnieje.



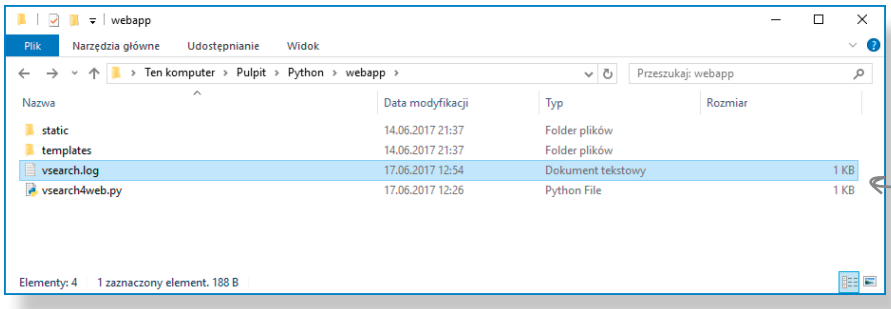


Trzecie (i ostatnie) wyszukiwanie

Dane są logowane (w tle)

Za każdym razem, gdy formularz HTML jest używany do wprowadzania danych do aplikacji WWW, funkcja `log_request` zachowuje szczegóły dotyczące żądania sieciowego i zapisuje wyniki w pliku logu. Bezpośrednio po pierwszym wyszukiwaniu w tym samym folderze, w którym znajduje się kod Twojej aplikacji, zostaje utworzony plik *vsearch.log*, tak jak zostało to pokazane poniżej.

W oknie naszego menedżera plików widoczna jest bieżąca zawartość folderu webapp.



Tutaj znajduje się plik vsearch.log.

Z pewnością kusi Cię, aby zobaczyć zawartość pliku *vsearch.log*, korzystając z edytora tekstowego. Ale co byłoby w tym *fajnego*? Jako że mamy do czynienia z aplikacją WWW, zapewnijmy dostęp do zalogowanych danych za pomocą samej aplikacji. Dzięki temu korzystając z niej, nigdy nie będziesz musiał opuszczać okna swojej przeglądarki internetowej. Utwórzmy zatem nowy URL, `/viewlog`, za pomocą którego będziemy mogli odwoływać się do strony wyświetlającej zawartość logu, gdy tylko zajdzie taka potrzeba.

Przeglądanie logu za pomocą naszej aplikacji WWW

W Twojej aplikacji WWW należy teraz dodać obsługę URL-a `/viewlog`. Gdy otrzyma ona żądanie o ten URL, powinna otworzyć plik *vsearch.log*, odczytać z niego wszystkie dane, a następnie przesłać je do oczekującej przeglądarki.

Większość rzeczy, które należy tu zrobić, już dobrze znasz. Zacznij od utworzenia pokazanego poniżej wiersza nowego dekoratora `@app.route` (odpowiedni kod dodamy na końcu pliku *vsearch4web.py*, tuż ponad wierszem *dunder name dunder main*).

```
@app.route('/viewlog')
```

Mamy tu całkiem nowy URL.

Po wybraniu odpowiedniego URL-a powinniśmy napisać dla niego funkcję. Nazwijmy ją `view_the_log`. Funkcja ta nie będzie przyjmowała żadnych argumentów, ale będzie zwracała do wywołującego kodu łańcuch znakowy, który powstanie w wyniku połączenia wszystkich wierszy danych odczytanych z pliku *vsearch.log*. Poniżej możesz zobaczyć, jak wygląda wiersz `def` naszej nowej funkcji.

```
def view_the_log() -> str:
```

Mamy też zupełnie nową funkcję, która (zgodnie z adnotacją) zwraca łańcuch znakowy.

Teraz trzeba napisać zestaw tej funkcji. Należy w nim otworzyć plik logu w trybie *do odczytu*. Jest to standardowy tryb funkcji `open`, dlatego w jej wywołaniu jako argument wystarczy podać nazwę pliku. Kontekstem, w którym przetwarzany jest nasz plik, możemy zarządzać za pomocą instrukcji `with`.

```
with open('vsearch.log') as log:
```

Otwórz plik logu do odczytu.

W obrębie zestawu instrukcji `with` powinniśmy odczytać wszystkie wiersze z pliku. Twoją pierwszą myślą mogłoby być zastosowanie pętli do przejścia przez plik wiersz po wierszu. Interpreter zapewnia jednak metodę `read`, która — gdy ją wywołać — zwraca *całą* zawartość pliku „za jednym zamachem". Poniżej znajdziesz pojedynczy wiersz kodu, który załatwia sprawę, tworząc nowy łańcuch znakowy o nazwie `contents`.

```
contents = log.read()
```

Odczytaj cały plik za „jednym zamachem" i przypisz jego zawartość do zmiennej (która otrzymała nazwę „contents").

Po odczytaniu pliku zestaw instrukcji `with` się kończy (zamykając plik), a Ty jesteś już gotów do wysłania danych do oczekującej przeglądarki internetowej. Jak widać poniżej, operacja ta jest bardzo prosta.

```
return contents
```

Weź listę wierszy przechowywanych w zmiennej contents i zwróć ją do kodu wywołującego.

Po zebraniu wszystkiego w całość otrzymamy dokładnie to, czego potrzebujemy, aby odpowiedzieć na żądanie dotyczące URL-a `/viewlog`.

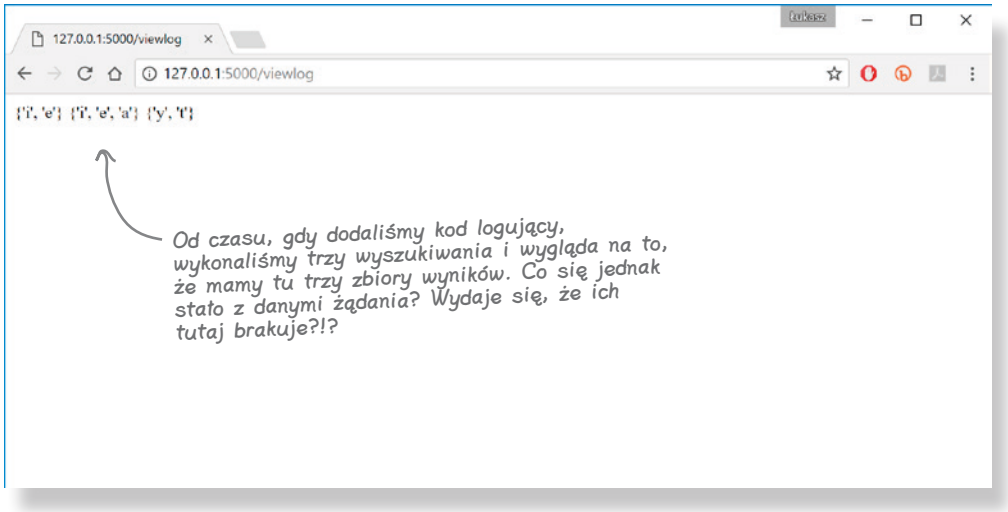
```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.read()
    return contents
```

To cały kod, którego potrzebujesz, aby zapewnić obsługę URL-a /viewlog.

Jazda próbna

Gdy dodasz i zapiszesz swój nowy kod, Twoja aplikacja WWW powinna zostać automatycznie przeładowana. Jeśli masz ochotę, możesz wykonać kilka operacji wyszukiwania, ale te, które przeprowadziłeś, są już zalogowane. Wszelkie nowe wyszukiwania, które wykonasz, zostaną dodane do pliku logu. Użyjmy URL-a `/viewlog`, aby sprawdzić, co zostało w nim zapisane. Wpisz ciąg ***http://127.0.0.1:5000/viewlog*** w polu adresu przeglądarki internetowej i naciśnij klawisz *Enter*.

Poniżej możesz zobaczyć, co ujrzeliśmy, korzystając z przeglądarki *Chrome* w systemie operacyjnym *Windows* (sprawdzaliśmy również przeglądarki *Safari* oraz *Firefox* i otrzymaliśmy te same wyniki).



Od czego zacząć, gdy coś niedobrego dzieje się z danymi wyjściowymi

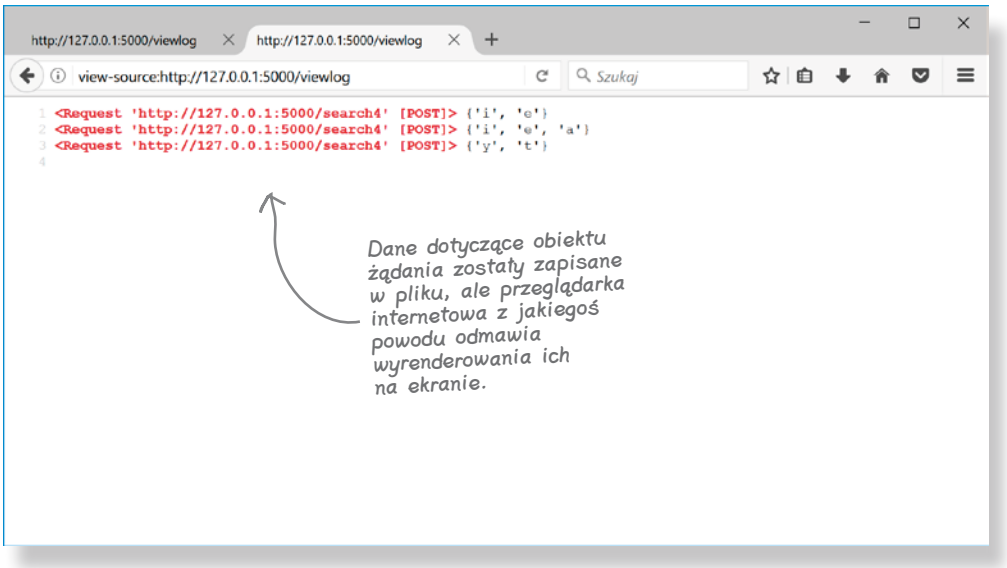
Jeśli Twoje dane wyjściowe nie wyglądają do końca tak, jak się tego spodziewasz (z czym *mamy* do czynienia powyżej), najlepiej zacząć od sprawdzenia, jakie dokładnie dane wysyła Ci aplikacja WWW. Ważna rzecz, którą należy odnotować, jest taka, że to, co widać na ekranie komputera, stanowi wykonany przez przeglądarkę *rendering* (lub interpretację) danych wygenerowanych przez aplikację WWW. Wszystkie ważniejsze przeglądarki internetowe umożliwiają przeglądanie surowych danych, które otrzymują (tzn. danych niepoddanych renderowaniu). Dane te znane są pod nazwą **źródło** (ang. *source*) strony, a ich przeglądanie może okazać się bardzo pomocne przy debugowaniu, jak również stanowić świetny pierwszy krok ku zrozumieniu tego, co się tu właściwie dzieje.

Jeśli korzystasz z programu *Firefox* lub *Chrome*, kliknij okno przeglądarki prawym przyciskiem myszy i wybierz z menu kontekstowego polecenie *Wyświetl źródło strony*, aby ujrzeć surowe dane przysłane przez aplikację WWW. Jeśli używasz *Safari*, musisz najpierw włączyć w nim opcje programistyczne. W tym celu otwórz okno preferencji *Safari*, przejdź do karty *Zaawansowane* i zaznacz widoczne w jej dolnej części pole wyboru *Pokazuj menu Programowanie na pasku menu*. Po zrobieniu tego możesz powrócić do okna przeglądarki, kliknąć je prawym przyciskiem myszy i wybrać z menu kontekstowego polecenie *Pokaż kod źródłowy strony*. Zrób to teraz i przyjrzyj się surowym danym, a następnie porównaj to z tym, co zobaczyliśmy my (a co pokazane zostało na następnej stronie).

Sprawdzanie surowych danych za pomocą przeglądarki źródeł strony

Pamiętaj, że funkcja `log_request` zapisuje dwa elementy danych dla każdego żądania, które loguje: obiekt tego żądania oraz wyniki wywołania funkcji `search4letters`. Jednak gdy oglądasz log (korzystając z URL-a `/viewlog`), widzisz jedynie dane związane z wynikami. Czy przejrzenie źródła strony (tzn. surowych danych zwróconych przez aplikację WWW) da nam jakąś wskazówkę na temat tego, co stało się z obiektem żądania?

Poniżej możesz sprawdzić, co zobaczyliśmy, gdy użyliśmy przeglądarki *Firefox*, aby przejrzeć surowe dane. Fakt, że każdy z obiektów żądania jest tu wyświetlony na czerwono, to kolejna wskazówka, że coś jest nie tak z naszymi danymi logu.



Powód, dla którego dane żądania nie są renderowane na ekranie, jest dość prozaiczny, a to, że *Firefox* zaznaczył te dane na czerwono, pomaga zrozumieć, co się tu dzieje. Okazuje się, że z samymi danymi żądania wszystko jest w jak najlepszym porządku. Wydaje się jednak, że dane umieszczone w nawiasach trójkątnych (tzn. pomiędzy znakami < oraz >) w jakiś sposób wyprowadzają przeglądarki z równowagi. Gdy napotykają one otwierający nawias trójkątny, traktują wszystko, co się po nim znajduje — aż do odpowiedniego nawiasu zamykającego — jako znacznik HTML. Z uwagi na to, że `<Request>` nie jest prawidłowym znacznikiem języka HTML, nowoczesne przeglądarki po prostu ignorują ten element i odmawiają renderowania jakiegokolwiek tekstu, który znajduje się w nawiasie. Z taką sytuacją mamy tu do czynienia, i to właśnie stanowi wyjaśnienie znikających danych żądań. Nadal jednak chcielibyśmy widzieć te dane, gdy przeglądamy stronę dostępną pod URL-em `/viewlog`.

Potrzebujemy tu jakiegoś sposobu, aby poinformować przeglądarkę, że nawiasów trójkątnych zawierających obiekt żądania nie powinna uważać za znacznik HTML, lecz za zwykły tekst. Mamy szczęście, ponieważ framework Flask zapewnia funkcję, która może nam tu pomóc.

Czas na użycie sekwencji ucieczki (w stosunku do Twoich danych)

Gdy powstawał język HTML, jego projektanci wiedzieli, że niektórzy twórcy stron internetowych będą chcieli wyświetlać znaki nawiasów trójkątnych (a także inne symbole mające specjalne znaczenie w HTML-u). W związku z tym wymyślili tzw. *escaping* (inaczej: użycie sekwencji ucieczki), czyli sposób kodowania znaków specjalnych HTML-a, dzięki któremu mogą one pojawiać się w stronach internetowych, lecz nie być przy tym interpretowane jako kod języka HTML. Zdefiniowano w tym celu szereg translacji, po jednej dla każdego znaku specjalnego. Idea jest prosta: znak specjalny, taki jak <, jest zdefiniowany jako `<`, natomiast znak > — jako `>`. Jeśli wyślesz te translacje *zamiast* surowych danych, Twoja przeglądarka zrobi to, co należy: wyświetli znaki < oraz >, zamiast je ignorować, a także pokaże tekst znajdujący się pomiędzy tymi znakami.

Framework Flask zawiera funkcję o nazwie `escape` (która tak naprawdę została odziedziczona po silniku Jinja2). Jeśli poda się jej surowe dane, tłumaczy je ona do postaci dostosowanej do wymagań kodu HTML. Poeksperymentujmy trochę z funkcją `escape` w wierszu poleceń Pythona, aby przekonać się, jak działa.

Porady maniaka

Zapewniany przez framework Flask obiekt **Markup** to tekst, który został sformatowany w taki sposób, aby był bezpieczny w kontekście użycia w kodzie HTML lub XML. Markup dziedziczy po wbudowanym łańcuchu znakowym Pythona o nazwie `unicode` i może być stosowany wszędzie tam, gdzie korzysta się ze zwykłych łańcuchów.

Zacznij od zaimportowania funkcji `escape` z modułu `flask`, a następnie wywołaj ją z argumentem będącym łańcuchem znakowym, który nie zawiera żadnych znaków specjalnych.

Zaimportuj funkcję.

```
>>> from flask import escape
>>> escape('To jest Request')
Markup('To jest Request')
```

Użyj funkcji escape w stosunku do zwykłego łańcucha znakowego.

Brak zmian

Wartość zwracana przez funkcję `escape` jest *obiektem Markup* (obiektem znacznikowym), który pod każdym względem zachowuje się tak, jak zwykły łańcuch znakowy. Jeśli przekażesz tej funkcji łańcuch zawierający jakiekolwiek znaki specjalne HTML-a, dokona ona jego translacji, tak jak zostało to przedstawione poniżej.

```
>>> escape('To jest <Request>')
Markup('To jest &lt;Request&gt;')
```

Użyj funkcji escape w stosunku do łańcucha znakowego zawierającego jakieś znaki specjalne.

Znaki specjalne zostały zastąpione odpowiednimi sekwencjami (tzn. przetłumaczone).

Tak jak w poprzednim przykładzie (pokazanym powyżej), również ten obiekt znacznikowy możesz traktować w taki sposób, jakby był zwykłym łańcuchem znakowym.

Gdyby udało się nam jakoś wywołać funkcję `escape` względem danych przechowywanych w pliku logu, bylibyśmy w stanie rozwiązać problem, który mamy obecnie z niewyświetlaniem danych żądań. Nie powinno to być zbyt trudne, jako że plik logu jest odczytywany „za jednym zamachem” przez funkcję `view_the_log`, zanim zwróci go ona w postaci łańcucha znakowego, jak pokazano poniżej.

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.read()
    return contents
```

Tutaj przechowywane są nasze zalogowane dane (w postaci łańcucha znakowego).

Aby rozwiązać nasz problem, musimy tylko wywołać funkcję `escape`, podając jej jako argument zmienną `contents`.

Przeglądanie pełnego logu w Twojej aplikacji WWW

Zmiana, której trzeba dokonać w Twoim kodzie, jest wręcz trywialna, ale robi naprawdę dużą różnicę. Dodaj funkcję `escape` do listy elementów importowanych z modułu `flask` (znajdującej się na początku kodu Twojego programu), a następnie wywołaj tę funkcję na rzecz łańcucha znakowego zwróconego przez metodę `read`.

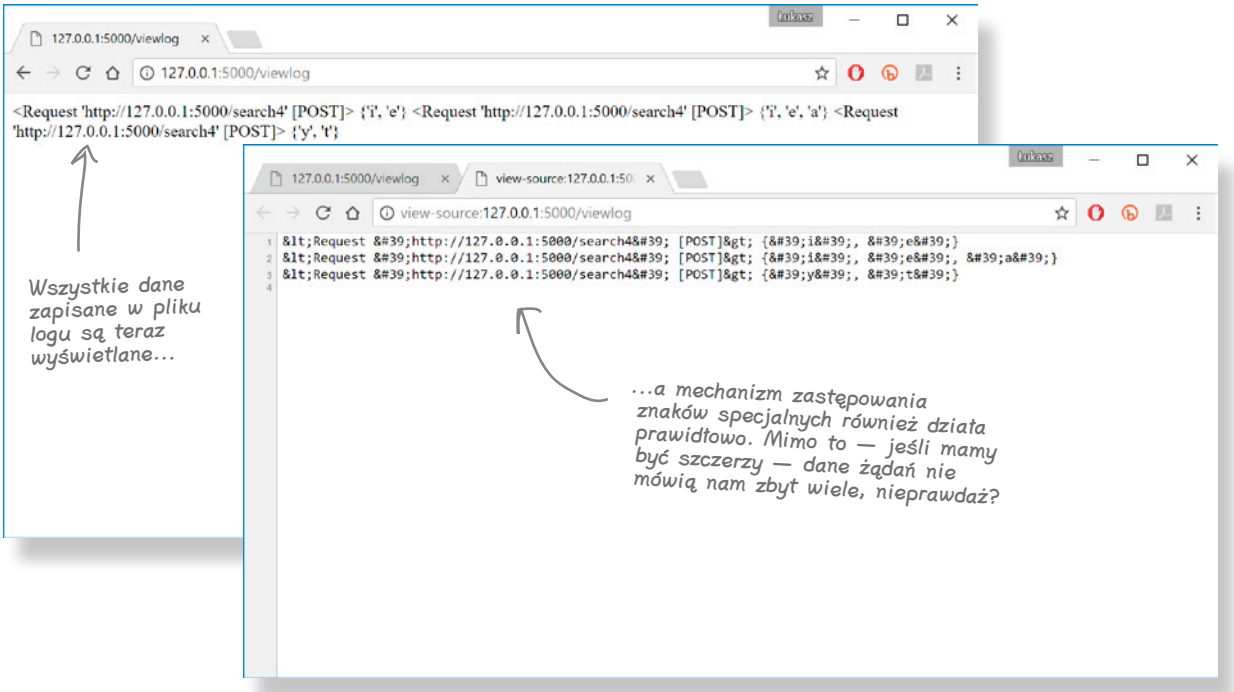
```
from flask import Flask, render_template, request, escape
     ...
@app.route('/viewlog')
def view_the_log() -> str:
   with open('vsearch.log') as log:
      contents = log.read()
   return escape(contents)
```

Dodaj to do listy importowanych elementów.

Wywołaj funkcję escape, podając jako jej argument zwrócony łańcuch znakowy.

Jazda próbna

Zmodyfikuj swój program w taki sposób, aby importował i wywoływał funkcję `escape`, jak zostało to przedstawione powyżej, a potem zapisz swój kod (aby Twoja aplikacja WWW została przeładowana). Następnie przeładuj w przeglądarce internetowej stronę wskazywaną przez URL `/viewlog`. Wszystkie dane pochodzące z logu powinny teraz pojawić się na ekranie. Koniecznie spójrz w źródło HTML, aby potwierdzić, że zastosowane rozwiązanie działa. Poniżej możesz się przekonać, co zobaczyliśmy, gdy przetestowaliśmy tę wersję naszej aplikacji WWW za pomocą przeglądarki *Chrome*.



Wszystkie dane zapisane w pliku logu są teraz wyświetlane...

...a mechanizm zastępowania znaków specjalnych również działa prawidłowo. Mimo to — jeśli mamy być szczerzy — dane żądań nie mówią nam zbyt wiele, nieprawdaż?

Więcej informacji na temat obiektu żądania

Znajdujące się w pliku logu dane dotyczące żądań sieciowych nie są w gruncie rzeczy aż tak użyteczne. Poniżej możesz znaleźć przykład tego, co jest w tej chwili logowane. Choć każdy zalogowany wynik jest inny, to każde zalogowane żądanie sieciowe jest *dokładnie* takie samo.

Każde zalogowane żądanie sieciowe jest takie samo.

```
<Request 'http://127.0.0.1:5000/search4' [POST]> {'i', 'e'}
<Request 'http://127.0.0.1:5000/search4' [POST]> {'i', 'e', 'a'}
<Request 'http://127.0.0.1:5000/search4' [POST]> {'y', 't'}
```

Każdy zalogowany wynik jest inny.

Logujemy żądania sieciowe na poziomie obiektu, ale tak naprawdę powinniśmy zaglądać do *wnętrza* żądań i logować pewne dane, które zawierają. Jak przekonałeś się już wcześniej w tej książce, gdy trzeba dowiedzieć się, co zawiera jakaś rzecz w Pythonie, należy podać nazwę tej rzeczy jako argument wbudowanej funkcji `dir`. Umożliwia to wyświetlenie listy metod i atrybutów danego obiektu.

Wprowadźmy niewielką zmianę w funkcji `log_request`, aby logować dane wyjściowe pochodzące z wywołania funkcji `dir` na rzecz obiektu żądania. Nie jest to wielka zmiana. Zamiast przekazywać zmienną `req` jako pierwszy argument funkcji `print`, przekażmy jej rzutowany na łańcuch tekstowy wynik wywołania funkcji `dir(req)`. Oto nowa wersja funkcji `log_request`, w której wyróżniona została wspomniana zmiana.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(str(dir(req)), res, file=log)
```

Wywołujemy funkcję dir, podając jej jako argument zmienną req, w wyniku czego zwracana jest lista, po czym przekształcamy tę listę w łańcuch znakowy, przekazując ją jako argument funkcji str. Otrzymany w efekcie tego łańcuch jest następnie zapisywany w pliku logu wraz z wartością zmiennej res.

Ćwiczenie

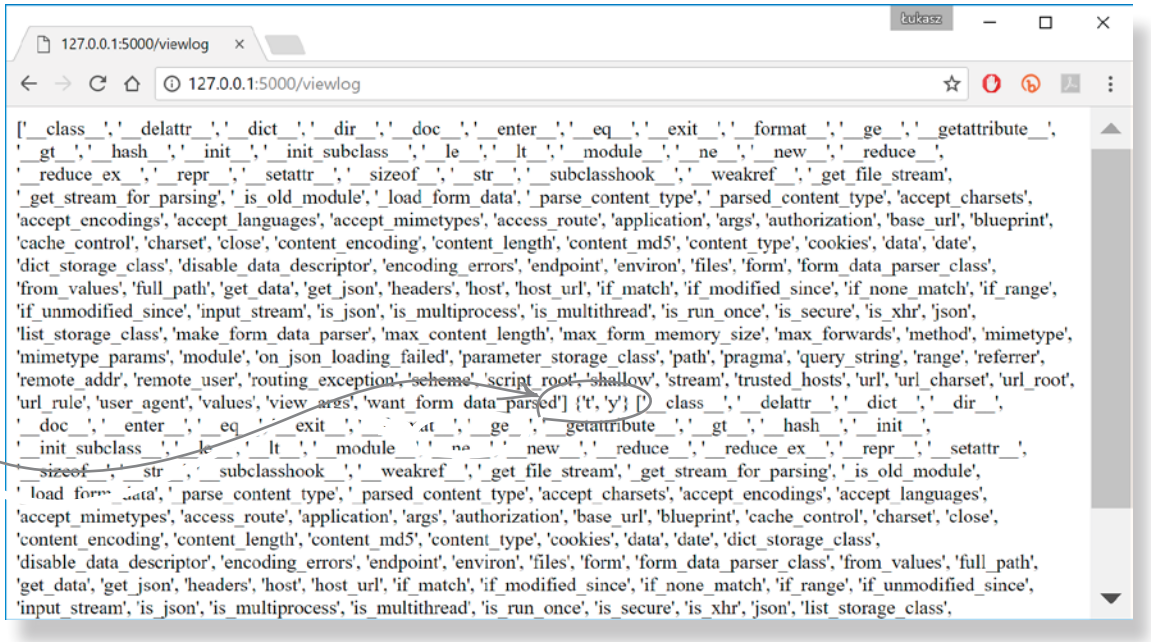
Wypróbujmy ten nowy kod logujący, aby przekonać się, co się dzięki niemu zmieni. Wykonaj następujące kroki:

1. Zmień swoją kopię funkcji `log_request` w taki sposób, aby wyglądała tak jak nasza.
2. Zapisz plik *vsearch4web.py*, aby ponownie uruchomić swoją aplikację WWW.
3. Znajdź i usuń bieżącą wersję pliku *vsearch.log*.
4. Użyj swojej przeglądarki internetowej, aby wykonać trzy nowe wyszukiwania.
5. Zobacz nowo utworzony log, korzystając z URL-a `/viewlog`.

Przyjrzyj się teraz dokładnie temu, co pojawiło się w oknie Twojej przeglądarki. Czy to, co w tej chwili w nim widzisz, może Ci w jakiś sposób pomóc?

Jazda próbna

Poniżej możesz się przekonać, co zobaczyliśmy, gdy wykonaliśmy pięć kroków przedstawionych pod koniec poprzedniej strony. Korzystaliśmy tu z programu *Chrome* (choć każda przeglądarka pokazuje te same dane).



Wszystko to wygląda dość chaotycznie. Przyjrzyj się jednak bliżej: tutaj widać wynik jednego z przeprowadzonych wyszukiwań.

Czym to wszystko zatem jest?

Wśród przedstawionych powyżej danych wyjściowych możesz znaleźć zalogowane wyniki działania funkcji wyszukującej znaki w podanej frazie. Reszta danych to wynik wywołania funkcji `dir` z argumentem będącym obiektem żądania. Jak widać, każdy z takich obiektów ma mnóstwo metod i atrybutów (będzie ich sporo nawet, gdy zignorujesz wszystkie *dundery* i *wondery*). Ewidentnie nie ma sensu logować *wszystkich* tych atrybutów.

Przyjrzeliśmy się wszystkim tym atrybutom i stwierdziliśmy, że trzy z nich wydają się na tyle istotne, że warto byłoby je logować. Są to:

- `req.form` — dane przesłane z formularza HTML aplikacji WWW;
- `req.remote_addr` — adres IP komputera, na którym działa używana przeglądarka internetowa;
- `req.user_agent` — tożsamość przeglądarki przesyłającej dane.

Zmieńmy kod funkcji `log_request` w taki sposób, aby logowała tylko te trzy szczególne fragmenty danych, oczywiście poza wynikami wywołań funkcji `search4letters`.

Logowanie wybranych atrybutów żądania sieciowego

Teraz, gdy masz już wybrane cztery elementy danych do logowania — szczegóły dotyczące formularza, adres IP klienta, tożsamość przeglądarki oraz wynik wywołania funkcji `search4letters` — pierwsze podejście do modyfikacji funkcji `log_request` może skutkować otrzymaniem kodu podobnego do przedstawionego poniżej, w którym każdy element danych jest logowany za pomocą osobnego wywołania funkcji `print`.

```
def log_request(req:'flask_request', res:str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, file=log)
        print(req.remote_addr, file=log)
        print(req.user_agent, file=log)
        print(res, file=log)
```

Zaloguj każdy element danych, korzystając z oddzielnego wywołania funkcji print.

Przedstawiony kod działa, ale wiąże się z nim pewien problem. Polega on na tym, że każde wywołanie funkcji `print` standardowo dokłada znak nowego wiersza, co oznacza, że na każde żądanie sieciowe przypadają aż **cztery** wiersze danych w pliku logu. Poniżej możesz zobaczyć, jak wyglądałyby te dane, gdybyśmy skorzystali z powyższego kodu.

Dla każdego adresu IP klienta występuje tu jeden wiersz danych.

Dane w postaci, w jakiej zostały wprowadzone w formularzu HTML, pojawiają się we własnym wierszu. Tak na marginesie: ImmutableMultiDict to specyficzny dla frameworka Flask rodzaj słownika Pythona (działa on w ten sam sposób, co standardowy słownik).

```
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])
127.0.0.1
Mozilla/5.0 (Windows NT 10.0; Win64; x64) ... Safari/537.36
{'e', 'i'}
ImmutableMultiDict([('phrase', 'autostopowicz'), ('letters', 'aeiou')])
127.0.0.1
Mozilla/5.0 (Windows NT 10.0; Win64; x64) ... Safari/537.36
{'u', 'o', 'a', 'i'}
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])
127.0.0.1
Mozilla/5.0 (Windows NT 10.0; Win64; x64) ... Safari/537.36
{'t', 'y'}
```

Informacje na temat przeglądarki zajmują własny wiersz.

Wyniki wywołania funkcji search4letters są tu wyraźnie pokazane (każdy w osobnym wierszu).

Właściwie nie ma nic złego w takim sposobie gromadzenia logów (zwłaszcza że zalogowane tą metodą dane łatwo się czyta). Zastanów się jednak, co musiałbyś zrobić, aby wczytać te dane do programu: każde zalogowane żądanie sieciowe wymagałoby aż **czterech** operacji odczytu z pliku logu, po jednej na każdy wiersz danych. I to pomimo faktu, że te cztery wiersze danych opisują tak naprawdę *pojedyncze* żądanie. Podejście to nie sprawdzi się w związku z tym najlepiej, ponieważ jest po prostu nieekonomiczne. Byłoby o wiele lepiej, gdyby nasz kod logował tylko **jeden** wiersz danych dla każdego żądania sieciowego.

Loguj pojedynczy wiersz rozdzielonych danych

Lepszą strategią logowania może być zapisywanie naszych czterech elementów danych w postaci jednego wiersza, przy czym poszczególne elementy powinny być rozdzielone odpowiednio dobranym separatorem.

Wybranie właściwego separatora może się okazać dość trudne, ponieważ trzeba uważać, aby nie użyć w tej roli znaku, który może występować w danych, jakie się loguje. Korzystanie ze spacji jako separatora właściwie mija się z celem (ponieważ logowane dane zawierają mnóstwo takich znaków), nawet zastosowanie dwukropka (`:`), przecinka (`,`) czy średnika (`;`) może być problemem, gdy weźmie się pod uwagę dane, które mają być logowane. Przedyskutowaliśmy sprawę z programistami pracującymi w laboratorium *Rusz głową!* i zasugerowali oni użycie w tej roli pionowej kreski (`|`). Jest to znak, który można dość łatwo zauważyć i raczej nie będzie występował w danych, które zamierzamy tu logować. Spróbujmy wdrożyć ten pomysł w życie i zobaczmy, jak sprawdzi się w praktyce.

Porady maniaka

O **separatorze** możesz myśleć jako o sekwencji składającej się z jednego znaku lub większej ich liczby, która to sekwencja odgrywa rolę elementu rozdzielającego poszczególne składniki wiersza tekstu. Klasycznym przykładem może tu być znak przecinka (`,`), wykorzystywany w plikach CSV.

Jak miałeś okazję już wcześniej się przekonać, możemy dostosować działanie funkcji `print`, podając jej dodatkowe argumenty. Oprócz argumentu `file` przyjmuje ona również argument `end`, za pomocą którego można określić własny *znak zakończenia wiersza*, alternatywny wobec standardowego znaku nowego wiersza.

Zmodyfikujmy funkcję `log_request` w taki sposób, aby w roli wartości zakończenia wiersza wykorzystywany był w niej znak pionowej kreski zamiast znaku nowego wiersza.

```
def log_request(req:'flask_request', res:str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, file=log, end='|')
        print(req.remote_addr, file=log, end='|')
        print(req.user_agent, file=log, end='|')
        print(res, file=log)
```

Każde z tych wywołań funkcji print korzysta z pionowej kreski zamiast ze standardowego znaku nowego wiersza.

Rozwiązanie to działa zgodnie z oczekiwaniami. Każde żądanie sieciowe powoduje teraz wygenerowanie pojedynczego wiersza logowanych danych, a poszczególne elementy rozdzielone są za pomocą pionowej kreski. Poniżej możesz zobaczyć, jak prezentują się dane w naszym pliku logu, gdy użyliśmy tej zmienionej wersji funkcji `log_request`.

Każde żądanie sieciowe ma swój własny wiersz w pliku logów (użyliśmy tu zawijania tekstu, aby zmieścić się na tej stronie).

```
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'i', 'e'}
ImmutableMultiDict([('phrase', 'autostopowicz'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0 (Windows
NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'o', 'a',
'i', 'u'}
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'t', 'y'}
```

Czy udało Ci się zauważyć pionowe kreski użyte w roli separatorów? Widać tu trzy takie znaki, co oznacza, że w każdym wierszu zalogowaliśmy cztery elementy danych.

Wysłaliśmy trzy żądania sieciowe, dlatego mamy również trzy wiersze danych w pliku logu.

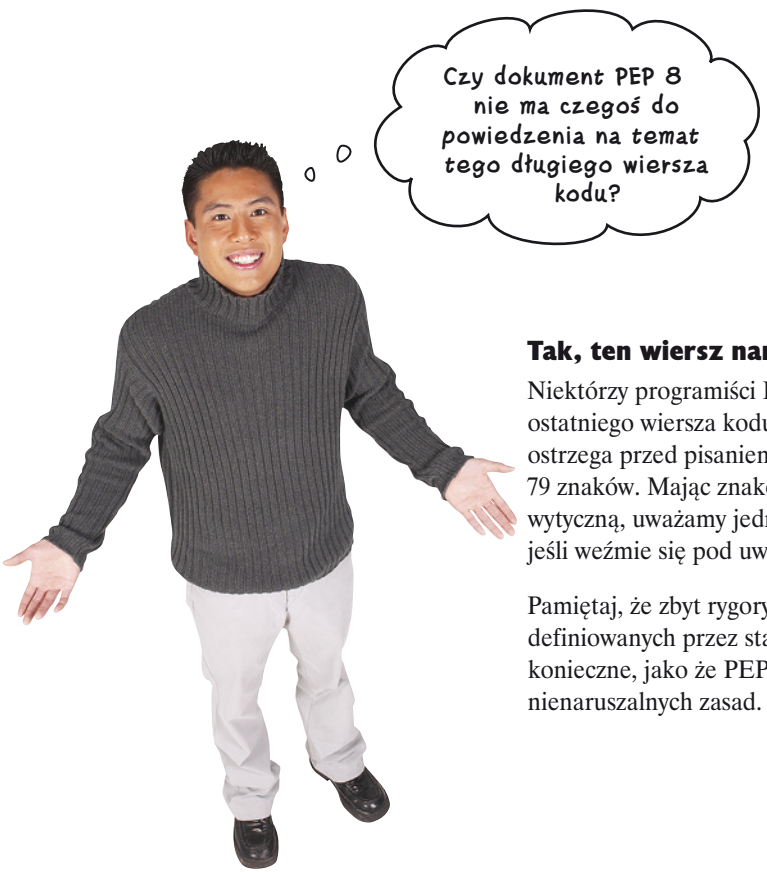
Ostatnia zmiana w naszym kodzie logującym

Praca z przesadnie rozwlekłym kodem przyprawia o ból głowy wielu programistów Pythona. Nasza ostatnia wersja funkcji `log_request` działa doskonale, ale nie jest tak zwięzła, jak mogłaby być. Przesadą wydaje się zwłaszcza zapisywanie poszczególnych elementów logowanych danych za pomocą osobnych wywołań funkcji `print`.

Funkcja `print` ma jeszcze jeden argument opcjonalny, noszący nazwę `sep`, za pomocą którego można określić wartość separującą używaną przy zapisywaniu wielu wartości za pomocą pojedynczego wywołania tej funkcji. Standardowo argumentowi temu przypisany jest pojedynczy znak spacji, ale możesz tu zastosować dowolną wartość, jaka przyjdzie Ci do głowy. W zaprezentowanym poniżej fragmencie kodu cztery wywołania funkcji `print` (przedstawione na poprzedniej stronie) zostały zastąpione pojedynczym wywołaniem tej funkcji, w którym wykorzystano możliwości oferowane przez argument `sep`, przypisując mu znak pionowej kreski. Pozwala to uniknąć konieczności podawania wartości argumentu `end` określającej znak zakończenia wiersza, dzięki czemu można się było również pozbyć z tego kodu wszystkich wystąpień tego argumentu.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```

Tylko jedno wywołanie funkcji print zamiast czterech



Tak, ten wiersz narusza wytyczne standardu PEP 8.

Niektórzy programiści Pythona skrzywią się na widok tego ostatniego wiersza kodu, jako że standard **PEP 8** wyraźnie ostrzega przed pisaniem wierszy o długości przekraczającej 79 znaków. Mając znaków 80, nasz wiersz kodu *trochę* nagina tę wytyczną, uważamy jednak, że stanowi rozsądny kompromis, jeśli weźmie się pod uwagę, co tu robimy.

Pamiętaj, że zbyt rygorystyczne trzymanie się wytycznych definiowanych przez standard PEP 8 nie jest bezwzględnie konieczne, jako że PEP 8 to tylko *przewodnik stylu*, a nie zbiór nienaruszalnych zasad. Wydaje nam się, że możemy ruszać dalej.

Ćwiczenie

Przekonajmy się, czym się różnią logi wygenerowane za pomocą tego nowego kodu od wcześniejszych. Zmień funkcję `log_request` w taki sposób, aby wyglądała tak, jak przedstawiona poniżej.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```

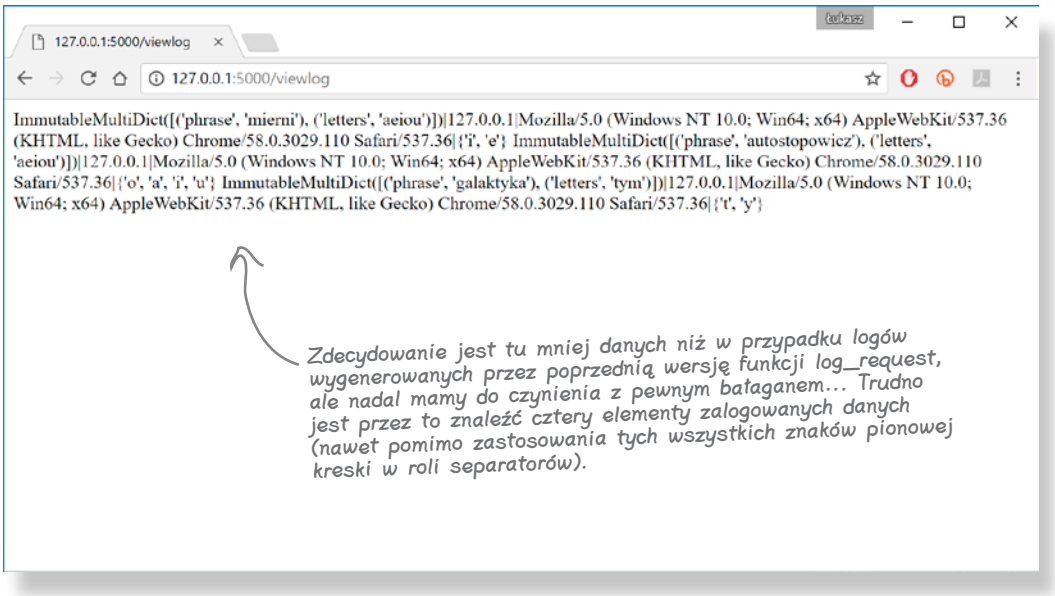
Następnie wykonaj cztery wymienione kroki:

1. Zapisz plik *vsearch4web.py*, aby ponownie uruchomić swoją aplikację WWW.
2. Znajdź i usuń bieżącą wersję pliku *vsearch.log*.
3. Użyj swojej przeglądarki internetowej, aby wykonać trzy nowe wyszukiwania.
4. Zobacz nowo utworzony log, korzystając z URL-a `/viewlog`.

Przyjrzyj się teraz dokładnie temu, co pojawiło się w oknie Twojej przeglądarki. Czy jest lepiej niż wcześniej?

Jazda próbna

Po wykonaniu czterech kroków wymienionych w powyższym ćwiczeniu uruchomiliśmy naszą nową testową wersję aplikacji, korzystając z przeglądarki *Chrome*. Oto co ukazało się naszym oczom.



Od surowych danych do czytelnych danych wyjściowych

Dane wyświetlone w oknie przeglądarki są w swojej *surowej formie*. Pamiętaj, że stosujemy wprawdzie escaping HTML, gdy odczytujemy dane z pliku logu, ale nie robimy już absolutnie nic więcej przed wysłaniem łańcucha znakowego do oczekującej przeglądarki internetowej. Nowoczesne przeglądarki odbiorą ten łańcuch, usuną z niego wszelkie niechciane białe znaki (takie jak dodatkowe spacje czy znaki nowego wiersza), a następnie wrzucą dane do swojego okna. Właśnie to stało się w czasie naszej *Jazdy próbnej*. Wszystkie zalogowane dane są widoczne, ale z pewnością trudno byłoby powiedzieć, że łatwo się je czyta. Moglibyśmy rozważyć jakieś dodatkowe działania na tych surowych danych (mające na celu zwiększenie ich czytelności), ale lepszym sposobem wygenerowania czytelnych danych wyjściowych może się okazać przetworzenie ich na postać tabelaryczną.

```
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'i', 'e'}
ImmutableMultiDict([('phrase', 'autostopowicz'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'o', 'a', 'i', 'u'}
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'t', 'y'}
```

Czy możemy wziąć te (nieczytelne) surowe dane...

...i przekształcić je w tabelę, która wygląda tak jak ta?

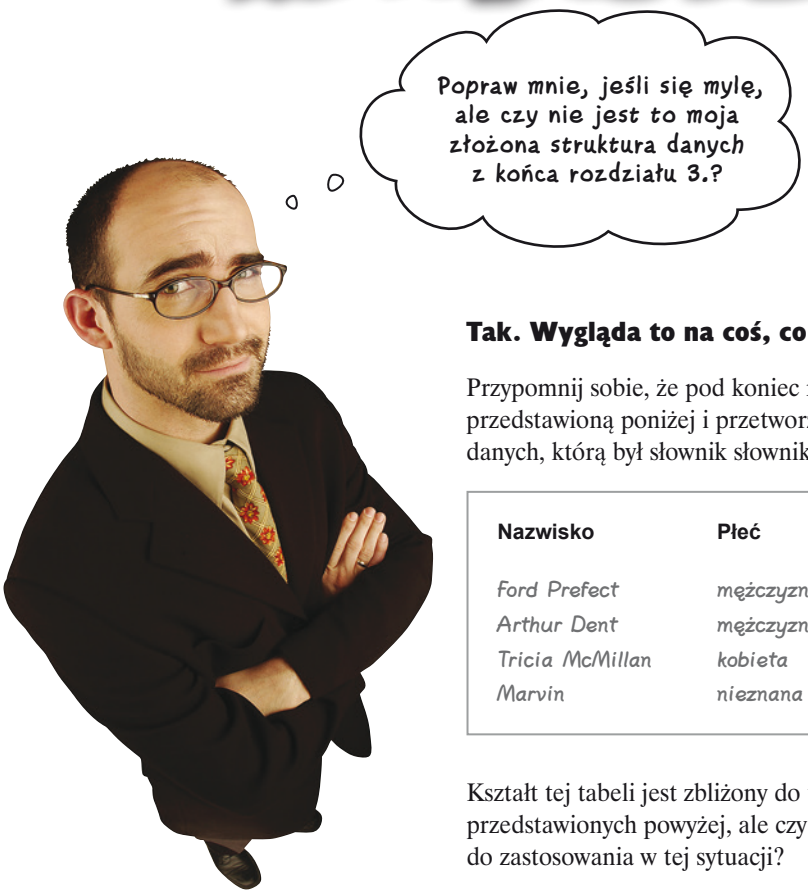
Dane z formularza	Adres klienta	Agent użytkownika	Wyniki
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])|	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'i', 'e'}
ImmutableMultiDict([('phrase', 'autostopowicz'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'o', 'a', 'i', 'u'}
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'t', 'y'}

Gdyby nasza aplikacja WWW była w stanie dokonać tej transformacji, wówczas *każdy* mógłby zobaczyć dane logu w swojej przeglądarce internetowej i łatwiej należycie je zrozumieć.

Czy coś Ci to przypomina?

Spójrz jeszcze raz na to, co próbujesz osiągnąć. Aby zaoszczędzić trochę miejsca, pokażemy tu tylko górną część tabeli przedstawionej w całości na poprzedniej stronie. Czy to, co masz zamiar wygenerować, przypomina Ci coś znanego z wcześniejszej części niniejszej książki?

Dane z formularza	Adres klienta	Agent użytkownika	Wyniki
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])|	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'i', 'e'}



Tak. Wygląda to na coś, co mieliśmy już okazję widzieć.

Przypomnij sobie, że pod koniec rozdziału 3. wzięliśmy tabelę danych przedstawioną poniżej i przetworzyliśmy ją w złożoną strukturę danych, którą był słownik słowników.

Nazwisko	Płeć	Zawód	Planeta macierzysta
Ford Prefect	mężczyzna	badacz	Betelgeza Siedem
Arthur Dent	mężczyzna	pan kanapka	Ziemia
Tricia McMillan	kobieta	matematyczka	Ziemia
Marvin	nieznana	paranoiczny android	nieznana

Kształt tej tabeli jest zbliżony do tego, co mamy nadzieję wygenerować z danych przedstawionych powyżej, ale czy słownik słowników to struktura danych odpowiednia do zastosowania w tej sytuacji?

Użyć słownika słowników... czy czegoś innego?

Do tabeli danych przedstawionej w rozdziale 3. pasuje model słownika słowników, ponieważ umożliwił on nam szybkie zagłębienie się w strukturze danych i wydobycie odpowiednich danych. Gdy chciałeś na przykład poznać nazwę macierzystej planety Forda Prefecta, mogłeś łatwo odwołać się do właściwej informacji za pomocą konstrukcji przedstawionej poniżej.

```
people['Ford']['Planeta macierzysta']
```

Odwołaj się do danych Forda...

...a następnie wydobądź wartość związaną z kluczem 'Planeta macierzysta'.

Jeśli chodzi o losowy dostęp do danych, nic nie jest w stanie pokonać słownika słowników. Ale czy tego właśnie potrzebujemy w przypadku naszych zalogowanych danych?

Zastanówmy się, co mamy w tej chwili.

Przyjrzyj się bliżej zalogowanym danym

Pamiętaj, że każdy zalogowany wiersz zawiera cztery elementy danych rozdzielone za pomocą pionowych kresek. Są to: dane pochodzące z formularza HTML, adres IP klienta, tożsamość przeglądarki internetowej oraz wynik wywołania funkcji `search4letters`.

Poniżej znajdziesz przykładowy wiersz danych pochodzący z naszego pliku *vsearch.log*. Został w nim wyróżniony każdy znak pionowej kreski.

Dane z formularza

Adres IP maszyny zdalnej

```
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'t', 'y'}
```

Łańcuch znakowy identyfikujący przeglądarkę internetową

Wyniki wywołania funkcji search4letters

Gdy zalogowane dane są odczytywane z pliku *vsearch.log*, lądują w Twoim kodzie w postaci listy *łańcuchów znakowych* dzięki temu, że użyliśmy metody `readlines`. Z racji tego, że raczej nie będziesz musiał losowo odwoływać się do poszczególnych elementów tych danych, przekształcanie ich w słownik słowników wydaje się dość kiepskim posunięciem. Musisz jednak być w stanie przetwarzać każdy wiersz *po kolei*, jak również przetwarzać *po kolei* każdy indywidualny element danych wchodzący w skład wiersza. Masz już listę łańcuchów znakowych, więc jesteś w połowie drogi do celu, ponieważ strukturę taką można łatwo przetwarzać za pomocą pętli `for`. Jednak każdy wiersz danych jest teraz jednym łańcuchem znakowym. Byłoby prościej przetwarzać każdy taki wiersz, gdyby zamiast tego stanowił listę elementów danych. Powstaje zatem pytanie: *czy da się skonwertować łańcuch znakowy na listę?*

Co zostało połączone, może zostać rozdzielone

Wiesz już, że listę łańcuchów znakowych możesz przekształcić w postać pojedynczego łańcucha za pomocą pewnej sztuczki, w której korzysta się z metody `join`. Poniżej pokazujemy ją raz jeszcze w wierszu poleceń powłoki.

Lista niezależnych łańcuchów znakowych

```
>>> names = ['Terry', 'John', 'Michael', 'Graham', 'Eric']
>>> pythons = '|'.join(names)
>>> pythons
'Terry|John|Michael|Graham|Eric'
```

Sztuczka z metodą join

Pojedynczy łańcuch znakowy utworzony z połączonych elementów listy names rozdzielonych znakami pionowej kreski

Dzięki tej sztuczce to, co było listą łańcuchów, stało się jednym łańcuchem, w którym każdy element listy jest oddzielony od następnego za pomocą pionowej kreski (w tym przypadku). Działanie to możesz odwrócić, korzystając z metody `split`, która jest wbudowana w każdy łańcuch znakowy w języku Python.

Weź łańcuch i podziel go na elementy listy, korzystając z podanego separatora.

```
>>> individuals = pythons.split('|')
>>> individuals
['Terry', 'John', 'Michael', 'Graham', 'Eric']
```

Teraz mamy z powrotem listę wyjściowych łańcuchów znakowych.

Przejście od listy łańcuchów znakowych do listy list

Gdy w swoim arsenale programistycznym masz już metodę `split`, wróćmy do danych przechowywanych w pliku logu i zastanówmy się, co powinno się tu wydarzyć, abyśmy zbliżyli się do celu. W tej chwili każdy wiersz odczytany z pliku *vsearch.log* ma postać łańcucha znakowego, tak jak zostało to pokazane poniżej.

Surowe dane

```
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'t', 'y'}
```

Działanie Twojego kodu polega teraz na odczytywaniu wszystkich wierszy z pliku *vsearch.log* i umieszczeniu ich na liście łańcuchów znakowych o nazwie `contents`. Poniżej zostały pokazane trzy ostatnie wiersze kodu funkcji `view_the_log`, która odczytuje dane z pliku i generuje z nich długi łańcuch znakowy.

```
...
with open('vsearch.log') as log:
    contents = log.readlines()
return escape(''.join(contents))
```

Otwórz plik logu...

...i odczytaj wszystkie znajdujące się w nim wiersze, a następnie umieść je na liście o nazwie „contents".

W ostatnim wierszu funkcji `view_the_log` bierze się wszystkie wiersze przechowywane na liście `contents` i łączy się je w jeden wielki łańcuch znakowy (przy użyciu metody `join`). Ten pojedynczy łańcuch zostaje następnie zwrócony do oczekującej przeglądarki internetowej.

Gdyby zmienna `contents` była listą *list* zamiast listą *łańcuchów znakowych*, moglibyśmy przetwarzać ją *po kolei* za pomocą pętli `for`. Dałoby się wówczas generować bardziej czytelne dane wyjściowe niż te, które możemy obecnie oglądać na ekranie.

Kiedy powinna się odbyć konwersja?

W tej chwili funkcja `view_the_log` odczytuje wszystkie dane z pliku logu i umieszcza je na liście łańcuchów znakowych (noszącej nazwę `contents`). Wolelibyśmy jednak raczej mieć tu listę list. Powstaje pytanie: kiedy jest „najlepszy czas", aby wykonać tę konwersję? Czy powinniśmy odczytać wszystkie dane i umieścić je na liście łańcuchów, a następnie przekształcać ją w listę list w kolejnym kroku, czy może raczej należałoby budować listę list już w trakcie odczytywania kolejnych wierszy danych?

To, że dane znajdują się już na liście `contents` (dzięki zastosowaniu metody `readlines`), nie powinno przesłonić nam faktu, że do tej chwili już raz przeszliśmy przez te dane za pomocą pętli. Wywołanie metody `readlines` może wydawać się z naszej perspektywy pojedynczą instrukcją, ale interpreter (wykonując tę metodę) *przechodzi* przez dane zapisane w pliku przy użyciu pętli. Jeśli ponownie będziemy je przetwarzać za pomocą pętli (aby przekształcić łańcuchy w listy), **podwoimy** liczbę operacji wykonywanych w pętli. Nie stanowi to wielkiego problemu, gdy mamy do czynienia z niewielką liczbą wpisów w logu... może jednak okazać się dość kłopotliwe, gdy nasz log znacznie zwiększy swoją wielkość. Najważniejszy wniosek z tych rozważań brzmi: *jeśli da się to osiągnąć za pomocą tylko jednej pętli, powinniśmy to tak właśnie zrobić!*

Przetwarzanie danych: co już wiemy

We wcześniejszej części tego rozdziału widziałeś pokazane poniżej trzy wiersze kodu Pythona, za pomocą których przetwarzane były wiersze danych przechowywane w pliku *todos.txt*.

Otwórz ten plik.

Przypisz strumień pliku do zmiennej.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

Przeprowadź jakieś przetwarzanie po jednym wierszu naraz.

Zapoznałeś się już również z metodą `split`, która przyjmuje argument będący łańcuchem znakowym i przekształca go w listę łańcuchów, korzystając przy tym z pewnego separatora (domyślnie jest nim znak spacji, jeśli nie zdefiniuje się tu innego). W przypadku naszych danych separatorem jest pionowa kreska. Załóżmy, że wiersz zalogowanych danych przechowywany jest w zmiennej o nazwie `line`. Zapisany w niej pojedynczy łańcuch znakowy możesz przekształcić w listę składającą się z czterech oddzielnych łańcuchów rozdzielonych znakiem pionowej kreski, korzystając z przedstawionej poniżej konstrukcji.

To jest nazwa nowo utworzonej listy.

Korzystamy z pionowej kreski jako separatora.

```
four_strings = line.split('|')
```

Używamy metody split, aby rozbić łańcuch znakowy na listę podłańcuchów.

Jako że nigdy nie możesz być pewien, że dane odczytywane z pliku logu nie zawierają żadnych znaków o specjalnym znaczeniu w języku HTML, nauczyłeś się też korzystać z funkcji `escape`. Funkcję tę zapewnia framework Flask, a jej zadaniem jest konwertowanie wszystkich znaków specjalnych HTML-a znajdujących się w łańcuchu na odpowiednie sekwencje ucieczki, tak jak zostało to pokazane poniżej.

```
>>> escape('To jest <Request>')
Markup('To jest &lt;Request&gt;')
```

Użyj funkcji escape w stosunku do łańcucha znakowego zawierającego jakieś znaki specjalne.

Ponadto już dawno temu, bo z rozdziału 2., dowiedziałeś się, że można utworzyć nową listę, przypisując zmiennej pustą listę (`[]`). Wiesz już też, że do końca istniejącej listy można dołączać wartości za pomocą wywołań metody `append`, a do ostatniego elementu listy odwołuje się za pomocą konstrukcji `[-1]`.

Utwórz nową, pustą listę o nazwie „names".

```
>>> names = []
>>> names.append('Michael')
>>> names.append('John')
>>> names[-1]
'John'
```

Dodaj jakieś dane na końcu istniejącej listy.

Odwołaj się do ostatniego elementu na liście names.

Uzbrojony w tę wiedzę, przekonaj się, czy jesteś w stanie wykonać ćwiczenie zaprezentowane na następnej stronie.

Zaostrz ołówek

Poniżej znajduje się aktualny kod funkcji `view_the_log`.

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.readlines()
    return escape(''.join(contents))
```

Kod ten odczytuje dane z pliku logu i umieszcza je na liście łańcuchów znakowych. Twoim zadaniem jest zmodyfikowanie tego kodu w taki sposób, aby odczytywał dane i umieszczał je na liście list.

Powinieneś zapewnić, aby dane zapisywane na liście były odpowiednio przetworzone przy użyciu sekwencji ucieczki, ponieważ nie chcemy, aby jakiekolwiek specjalne znaki języka HTML cichcem pojawiły się na liście.

Zadbaj też o to, aby Twój nowy kod nadal zwracał łańcuch znakowy do oczekującej przeglądarki internetowej.

Zaczęliśmy za Ciebie, Ty uzupełnij brakujący kod.

Dwa pierwsze wiersze funkcji pozostają bez zmian.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
```

.....

.....

.....

.....

.....

.....

.....

.....

.....

Tutaj dodaj swój nowy kod.

Funkcja nadal zwraca łańcuch znakowy.

```
return str(contents)
```

Nie spiesz się tu zbytnio. Śmiało eksperymentuj w wierszu poleceń powłoki tak długo, jak długo uznasz za stosowne. Nie martw się też, jeśli zdarzy Ci się utknąć w martwym punkcie. Nie ma sprawy, jeśli postanowisz wtedy przejść na następną stronę i spojrzeć na gotowe rozwiązanie.

Zaostrz ołówek
Rozwiązanie

Poniżej znajduje się kod funkcji `view_the_log`.

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.readlines()
    return escape(''.join(contents))
```

Twoim zadaniem było zmodyfikowanie tego kodu w taki sposób, aby odczytywał dane i umieszczał je na liście list.

Miałeś zapewnić, aby zapisywane na liście dane były odpowiednio przetworzone przy użyciu sekwencji ucieczki, ponieważ nie chcemy, aby jakiekolwiek specjalne znaki języka HTML cichcem pojawiły się na liście.

Powinieneś był również zadbać o to, aby Twój nowy kod nadal zwracał łańcuch znakowy do oczekującej przeglądarki internetowej.

Zaczęliśmy za Ciebie, a Ty miałeś uzupełnić brakujący kod.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

Utwórz nową, pustą listę o nazwie „contents".

Otwórz plik logu i przypisz go do strumienia pliku o nazwie „log".

Przejdź za pomocą pętli przez każdy wiersz należący do strumienia pliku log.

Dołącz nową, pustą listę do listy contents.

Podziel bieżący wiersz (korzystając ze znaków pionowej kreski jako separatorów), a następnie przetwórz każdy z elementów należących do powstałej w wyniku tego listy split.

Czy pamiętałeś o użyciu funkcji escape?

Dołącz przetworzone dane do końca listy znajdującej się na końcu listy contents.

Nie martw się, jeśli przedstawiony poniżej wiersz kodu należący do pokazanej powyżej funkcji `view_the_log` przyprawia Cię o ból głowy.

```
contents[-1].append(escape(item))
```

Przeczytaj ten kod, zaczynając od środka, a kończąc na zewnątrz, i od prawej do lewej strony.

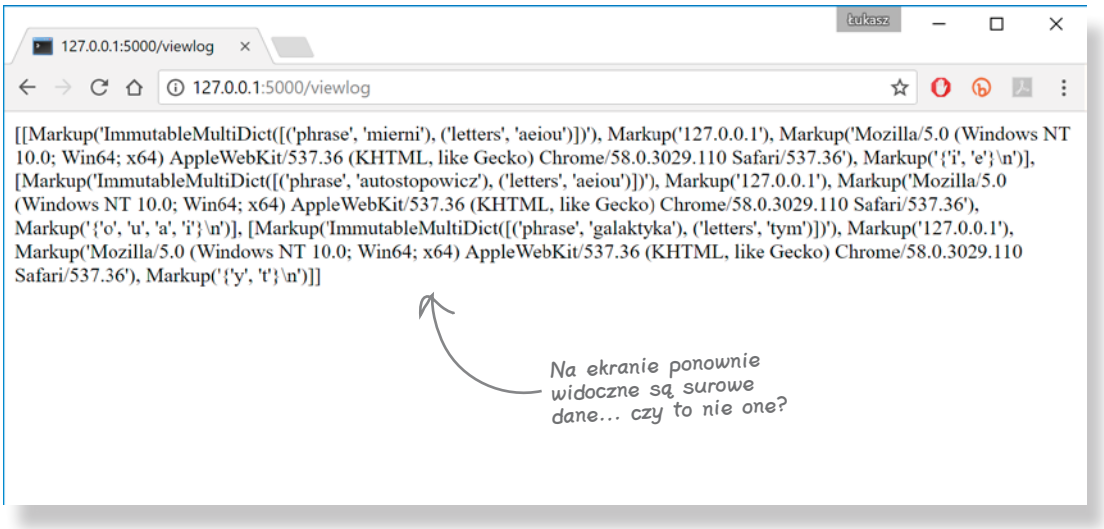
Sztuczka, która pozwala zrozumieć ten (początkowo dość odstręczający) wiersz kodu, polega na odczytaniu go od środka na zewnątrz i od prawej do lewej strony. Powinieneś zatem zacząć od zmiennej iteracyjnej `item` należącej do zestawu wewnętrznej pętli `for`, która to zmienna zostaje przekazana jako argument wywołania funkcji `escape`. Powstały w wyniku tego łańcuch znakowy jest następnie przypisywany do listy znajdującej się na końcu (`[-1]`) listy `contents`. Pamiętaj, że ta ostatnia jest *listą list*.

Jazda próbna

Wprowadź teraz zmiany w kodzie swojej funkcji `view_the_log`, aby przyjął on postać przedstawioną poniżej.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

Zapisz swój kod (powodując przeładowanie Twojej aplikacji WWW), a następnie przeładuj w przeglądarce stronę wskazywaną przez URL `/viewlog`. Poniżej możesz się przekonać, co zobaczyliśmy w naszej przeglądarce, gdy to zrobiliśmy.



Przyjrzyj się bliżej danym wyjściowym

Na pierwszy rzut oka dane wyjściowe wygenerowane przez tę nową wersję funkcji `view_the_log` wyglądają bardzo podobnie do tego, z czym mieliśmy do czynienia wcześniej. W rzeczywistości nie są jednak tym samym. Nowe dane mają postać listy list, a nie listy łańcuchów znakowych. Zmiana ta ma kolosalne znaczenie. Gdybyś mógł teraz przetworzyć dane z listy `contents`, korzystając z odpowiednio zaprojektowanego szablonu silnika Jinja2, powinieneś być w stanie znacznie zbliżyć się do celu, jakim jest tutaj otrzymanie czytelnych danych wyjściowych.

Generowanie czytelnych danych wyjściowych przy użyciu kodu HTML

Przypomnij sobie, że naszym celem jest wygenerowanie danych wyjściowych, które na ekranie będą się prezentowały lepiej niż surowe dane przedstawione na poprzedniej stronie. Pomoże nam w tym język HTML, zapewniający zestaw znaczników, które umożliwiają definiowanie zawartości tabel. Do znaczników tych należą: `<table>`, `<th>`, `<tr>` oraz `<td>`. Mając to na uwadze, przyjrzyjmy się raz jeszcze górnej części tabeli, którą zamierzamy uzyskać. Każdemu wierszowi logu odpowiada w niej wiersz podzielony na cztery kolumny (z których każda ma nagłówek opisujący jej przeznaczenie).

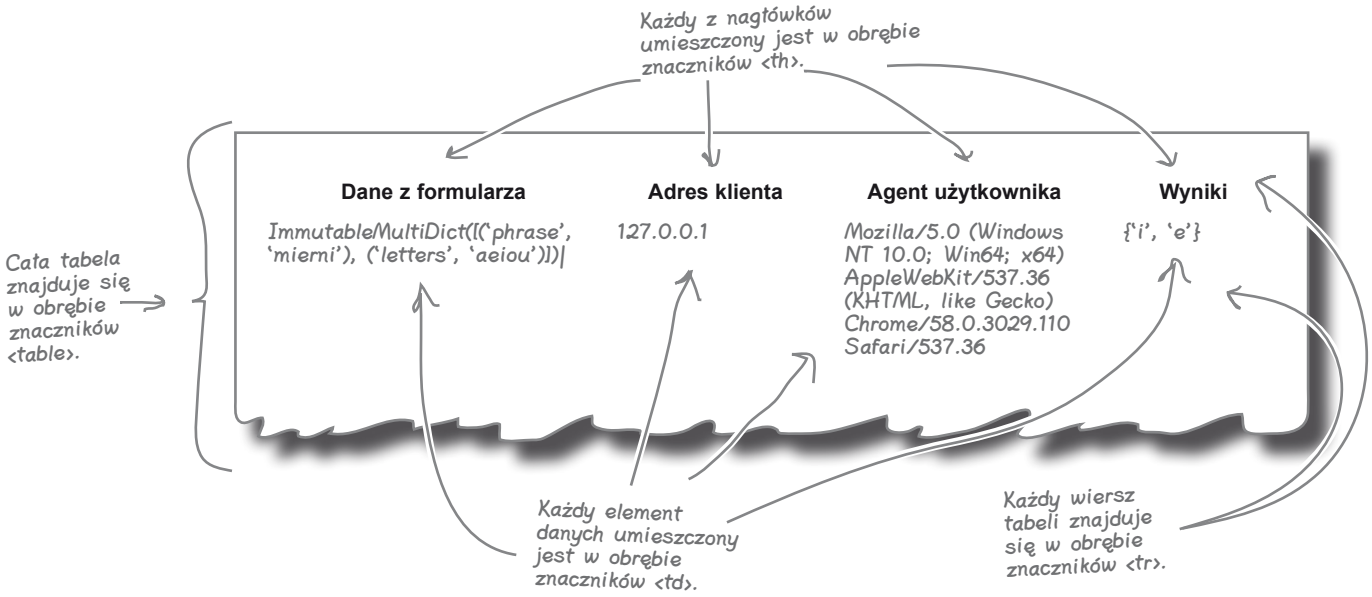
Całą tabelę mógłbyś umieścić w HTML-owym znaczniku `<table>`, a poszczególne wiersze danych objąć znacznikami `<tr>`. Nagłówki otrzymałyby w tym przypadku swoje znaczniki `<th>`, a każdy składnik surowych danych — znaczniki `<td>`, tak jak widać to poniżej.

Porady maniaka

Oto krótki przegląd znaczników języka HTML związanych z tabelami:

- `<table>` — tabela,
- `<tr>` — wiersz danych tabeli,
- `<th>` — nagłówek kolumny tabeli,
- `<td>` — element danych tabeli (komórka).

Każdy z wymienionych znaczników ma odpowiadający mu znacznik końcowy. Są to odpowiednio: `</table>`, `</tr>`, `</th>` oraz `</td>`.



Gdybyś kiedykolwiek stwierdził, że musisz wygenerować kod HTML (zwłaszcza taki, w którym występują znaczniki `<table>`), przypomnij sobie o silniku szablonów Jinja2. Powstał on przede wszystkim z myślą o generowaniu stron HTML i oferuje szereg podstawowych konstrukcji programistycznych (luźno wykorzystujących składnię języka Python), z których możesz korzystać, aby „automatyzować" wszelką związaną z wyświetlaniem danych logikę, której możesz potrzebować.

W poprzednim rozdziale przekonałeś się, jak obsługiwane przez silnik Jinja2 znaczniki `{{ i }}`, jak również znacznik `{% block %}` umożliwiają używanie zmiennych i bloków kodu HTML jako argumentów szablonów. Okazuje się, że znaczniki `{%` oraz `%}` mają znacznie bardziej ogólny charakter i mogą zawierać dowolną *instrukcję* silnika Jinja2, a jedną z obsługiwanych tu instrukcji jest konstrukcja pętli `for`. Na następnej stronie dowiesz się, jak wykorzystać tę konstrukcję, aby zbudować czytelne dane wyjściowe na podstawie listy list przechowywanej w zmiennej `contents`.

Osadzanie logiki wyświetlania w swoim szablonie

Poniżej znajdziesz nowy szablon, o nazwie *viewlog.html*, który zostanie użyty do przekształcenia surowych danych pochodzących z pliku logu w tabelę HTML. Szablon ten oczekuje listy list `contents` jako jednego ze swoich argumentów. Wyróżniliśmy te fragmenty szablonu, na których powinieneś się skupić. Zauważ, że zapewniana przez silnik Jinja2 konstrukcja pętli `for` jest bardzo podobna do tej, która jest stosowana w języku Python. Są jednak między nimi dwie istotne różnice:

Nie musisz tworzyć tego szablonu samodzielnie. Pobierz go spod adresu *ftp://ftp.helion.pl/przykłady/pytrg2.zip*.

- Nie ma tu potrzeby używania znaku dwukropka (`:`) na końcu wiersza zawierającego instrukcję `for` (ponieważ znacznik `%}` odgrywa rolę separatora).
- Zestaw pętli kończy się znacznikiem `{% endfor %}`, ponieważ silnik Jinja2 nie obsługuje wcięć (a więc potrzebny jest tutaj jakiś inny mechanizm).

Jak widać, pierwsza pętla `for` spodziewa się znaleźć swoje dane w zmiennej o nazwie `the_row_titles`, podczas gdy druga spodziewa się swoich danych w czymś o nazwie `the_data`. Trzecia pętla (zagnieżdżona w drugiej) oczekuje danych w postaci listy list.

Kod gotowy do użycia

Aby zapewnić spójny wygląd wszystkich stron, niniejszy szablon dziedziczy po tym samym szablonie bazowym, który był używany w całej naszej aplikacji WWW.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<table>
    <tr>
        {% for row_title in the_row_titles %}
            <th>{{row_title}}</th>
        {% endfor %}
    </tr>
    {% for log_row in the_data %}
        <tr>
            {% for item in log_row %}
                <td>{{item}}</td>
            {% endfor %}
        </tr>
    {% endfor %}
</table>

{% endblock %}
```

Opisowe tytuły (każdy umieszczony pomiędzy odpowiednimi znacznikami <th>) mają swój osobny wiersz (umieszczony w obrębie znaczników <tr>).

Cała tabela umieszczona jest w obrębie znaczników <table>.

Każdy element danych znajduje się pomiędzy znacznikami <td>, a każdy wiersz tabeli ma własny zestaw znaczników <tr>.

Pamiętaj, aby umieścić ten nowy szablon w folderze *templates* Twojej aplikacji WWW, zanim spróbujesz z niego skorzystać.

Produkcja czytelnych danych wyjściowych przy użyciu silnika Jinja2

Jako że szablon *viewlog.html* dziedziczy po szablonie *base.html*, musisz pamiętać, aby zapewnić wartość dla argumentu `the_title` oraz dostarczyć listę nagłówków kolumn (opisowych tytułów) za pomocą argumentu `the_row_titles`. Nie zapomnij też przypisać listy `contents` do argumentu `the_data`.

Funkcja `view_the_log` wygląda w tej chwili tak, jak zostało to pokazane poniżej.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

Obecnie zwracamy oczekującej przeglądarce łańcuch znakowy.

Powinieneś wywołać funkcję `render_template` w stosunku do pliku *viewlog.html* i przekazać jej odpowiednie wartości dla każdego z trzech argumentów, których oczekuje. Utwórzmy krotkę zawierającą opisowe tytuły i przypiszmy ją do argumentu `the_row_titles`, a następnie przypiszmy wartość zmiennej `contents` do argumentu `the_data`. Zapewnimy też właściwą wartość argumentu `the_title`, zanim wyrenderujemy nasz szablon.

Pamiętaj, że krotka to lista tylko do odczytu.

Mając to wszystko na uwadze, zmieńmy funkcję `view_the_log` tak, jak zostało to pokazane poniżej (wprowadzone przez nas zmiany zostały odpowiednio wyróżnione).

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Dane z formularza', 'Adres klienta', 'Agent użytkownika', 'Wyniki')
    return render_template('viewlog.html',
                           the_title='Widok logu',
                           the_row_titles=titles,
                           the_data=contents,)
```

Zmień adnotację w taki sposób, aby wskazywała, że (zamiast łańcucha znakowego) zwracany jest kod HTML.

Utwórz krotkę zawierającą opisowe tytuły.

Wywołaj funkcję render_template, przekazując jej wartości dla każdego argumentu naszego szablonu.

Rusz do boju i wprowadź przedstawione powyżej zmiany w swojej funkcji `view_the_log`, a następnie zapisz je, aby framework ponownie uruchomił Twoją aplikację WWW. Gdy będziesz gotów, przejrzyj log za pomocą swojej przeglądarki internetowej, korzystając z ciągu URL o postaci: *http://127.0.0.1:5000/viewlog*.

Jazda próbna

Poniżej możesz się przekonać, co ujrzeliśmy, gdy weszliśmy na stronę prezentującą logi naszej zaktualizowanej aplikacji WWW. Strona ta wygląda teraz tak samo jak nasze wszystkie inne strony, dzięki czemu mamy pewność, że aplikacja korzysta z odpowiedniego szablonu.

Jesteśmy całkiem zadowoleni z uzyskanego efektu (i mamy nadzieję, że Ty też jesteś), ponieważ jest on bardzo podobny do tego, co chcieliśmy osiągnąć, a dane wyjściowe są wyświetlane w bardzo czytelnej formie.

Widok logu

127.0.0.1:5000/viewlog

Widok logu

Dane z formularza	Adres klienta	Agent użytkownika	Wyniki
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'i', 'e'}
ImmutableMultiDict([('phrase', 'autostopowicz'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'o', 'u', 'a', 'i'}
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'y', 't'}

Dane wyjściowe są nie tylko czytelne, ale też prezentują się bardzo dobrze :)

Gdy wyświetlisz źródło przedstawionej powyżej strony (klikając prawym przyciskiem myszy okno przeglądarki i wybierając odpowiednie polecenie z menu kontekstowego), przekonasz się, że każdy element danych z logu otrzymał własny znacznik `<td>`, każdy wiersz danych ma swój znacznik `<tr>`, a cała tabela znajduje się w odpowiednich znacznikach `<table>`.

Bieżący stan kodu naszej aplikacji WWW

Zatrzymajmy się na chwilę i przejrzyjmy kod naszej aplikacji WWW. Dodanie kodu logującego (w postaci funkcji `log_request` oraz `view_the_log`) spowodowało rozrost naszej bazy kodu, ale całość programu nadal mieści się na jednej stronie. Poniżej znajdziesz kod programu *vsearch4web.py* widoczny w oknie edycji środowiska IDLE (które umożliwia Ci podziwianie kodu w pełni składniowej chwały kolorystycznej).

vsearch4web.py - C:\Users\lsuma\Desktop\Python\webapp\vsearch4web.py (3.6.0)

File Edit Format Run Options Window Help

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

app = Flask(__name__)

def log_request(req: 'flask_request', res: str) -> None:
   with open('vsearch.log', 'a') as log:
      print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   log_request(request, results)
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

@app.route('/viewlog')
def view_the_log() -> 'html':
   contents = []
   with open('vsearch.log') as log:
      for line in log:
         contents.append([])
         for item in line.split('|'):
            contents[-1].append(escape(item))
   titles = ('Dane z formularza', 'Adres klienta', 'Agent użytkownika', 'Wyniki')
   return render_template('viewlog.html',
                          the_title='Widok logu',
                          the_row_titles=titles,
                          the_data=contents,)

if __name__ == '__main__':
   app.run(debug=True)
```

Ln: 1 Col: 0

Zadawanie pytań na temat swoich danych

Możliwości naszej aplikacji WWW rozwijają się nieźle, ale czy zbliżyliśmy się do odpowiedzi na pytania zadane na samym początku tego rozdziału, tzn.: *Na ile żądań udało jej się odpowiedzieć? Jaka jest najczęściej używana lista liter? Z których adresów IP nadchodzą żądania? Z której przeglądarki internetowej najczęściej korzystają użytkownicy naszej aplikacji?*

Na ostatnie dwa z tych pytań w pewnym sensie udało się odpowiedzieć za pomocą danych wyjściowych wyświetlonych po przejściu pod URL `/viewlog`. Możesz stwierdzić, skąd napływają żądania (spójrz w dane zgromadzone w kolumnie **Adres klienta**), jak również sprawdzić, z których przeglądarek internetowych korzystali użytkownicy (popatrz na dane wyświetlone w kolumnie **Agent użytkownika**). Gdybyś jednak chciał obliczyć, która z popularnych przeglądarek jest najczęściej wykorzystywana przez osoby odwiedzające Twój serwis, nie byłoby to łatwe. Samo patrzenie w wyświetlone dane logu nie wystarczy; będziesz musiał przeprowadzić dodatkowe obliczenia.

Na pierwsze dwa pytania również nie da się łatwo odpowiedzieć. Powinno być jasne, że także tutaj niezbędne są dodatkowe obliczenia.

Pisz więcej kodu tylko wtedy, gdy musisz.

Gdyby wszystkim, co mamy do dyspozycji, był język Python, wówczas: tak, musielibyśmy napisać znacznie więcej kodu, aby odpowiedzieć na te pytania (i wszystkie inne, które mogłyby się pojawić). W końcu pisanie kodu Pythona jest niezłą zabawą, a język ten świetnie się też nadaje do manipulowania danymi. Napisanie dodatkowego kodu, dzięki któremu poznalibyśmy odpowiedzi na nasze pytania, wydaje się łatwizną, prawda?

Coż... Istnieją inne technologie, które znacznie upraszczają znajdowanie odpowiedzi na pytania tego rodzaju, jakie postawiliśmy, i uwalniają nas od konieczności pisania dodatkowego kodu Pythona. A konkretnie, gdybyśmy mogli zapisać dane logu w bazie danych, dałoby się skorzystać z możliwości oferowanych przez technologię zapytań bazodanowych, aby znaleźć odpowiedzi na niemal wszystkie pytania, jakie mogłyby się pojawić.

W następnym rozdziale dowiesz się, co trzeba zmienić w naszej aplikacji WWW, aby logowała ona swoje dane w bazie danych zamiast w pliku tekstowym.

Kod przykładów z rozdziału 6.

Pamiętaj, że obydwa te fragmenty robią dokładnie do samo, ale programiści Pythona wolą korzystać z tego kodu zamiast z tego.

```
with open('todos.txt') as tasks:
   for chore in tasks:
      print(chore, end='')
```

```
tasks = open('todos.txt')
for chore in tasks:
   print(chore, end='')
tasks.close()
```

To jest kod, który dodaliśmy do aplikacji WWW, aby zapewnić logowanie naszych żądań sieciowych w pliku tekstowym.

```
    ...

def log_request(req: 'flask_request', res: str) -> None:
   with open('vsearch.log', 'a') as log:
      print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')

    ...

@app.route('/viewlog')
def view_the_log() -> 'html':
   contents = []
   with open('vsearch.log') as log:
      for line in log:
         contents.append([])
         for item in line.split('|'):
            contents[-1].append(escape(item))
   titles = ('Dane z formularza', 'Adres klienta', 'Agent użytkownika', 'Wyniki')
   return render_template('viewlog.html',
                          the_title='Widok logu',
                          the_row_titles=titles,
                          the_data=contents,)

    ...
```

Nie prezentujemy tu całego kodu programu vsearch4web.py, lecz tylko nowe rzeczy, które się w nim pojawiły. (Cały kod programu znajdziesz dwie strony wcześniej).

Stosowanie DB-API języka Python

Przechowywanie danych w relacyjnej bazie danych jest wygodne.

W tym rozdziale dowiesz się, jak pisać kod umożliwiający korzystanie z popularnej bazy danych **MySQL** za pomocą ogólnego interfejsu o nazwie **DB-API**. API to (zapewniane standardowo przez każdą instalację języka Python) umożliwia tworzenie kodu, który można łatwo przenosić z jednego systemu bazy danych do innych, jeśli tylko systemy te wykorzystują język SQL. Chociaż będziemy stosować MySQL-a, nic nie stoi na przeszkodzie, abyś kod używający DB-API wykorzystywał do działań na innych relacyjnych bazach danych, niezależnie od tego, jaki konkretnie system wybierzesz. Sprawdźmy zatem, jak w języku Python wygląda obsługa relacyjnej bazy danych. W rozdziale tym nie znajdziesz wielu nowych informacji na temat samego Pythona, ale używanie tego języka do komunikacji z bazami danych jest **niezwykle istotną kwestią**, dlatego warto dobrze ją poznać.

Włączanie obsługi bazy danych w Twojej aplikacji WWW

Plan, który zamierzamy zrealizować w tym rozdziale, zakłada dojście do punktu, w którym będziesz mógł zmienić swoją aplikację WWW w taki sposób, aby przechowywała swoje logi w bazie danych zamiast w pliku tekstowym, jak robiła to w poprzednim rozdziale. Mamy nadzieję, że dzięki temu będziesz mógł następnie odpowiedzieć na pytania, które się w nim pojawiły, a więc: *Na ile żądań udało jej się odpowiedzieć? Jaka jest najczęściej używana lista liter? Z których adresów IP nadchodzą żądania? Z której przeglądarki internetowej najczęściej korzystają użytkownicy naszej aplikacji?*

Aby dostać się do tego miejsca, musimy jednak najpierw wybrać system bazy danych, z którego chcemy korzystać. Mamy tu mnóstwo opcji do wyboru i z łatwością zapełnilibyśmy kilkanaście stron samym tylko opisem paru alternatywnych technologii bazodanowych oraz szczegółami na temat mocnych i słabych stron każdej z nich. Nie zrobimy tego jednak, lecz wybierzemy po prostu popularny system i skorzystamy z bazy danych *MySQL*.

Wybrawszy bazę danych MySQL, przedstawmy cztery zadania, które zamierzamy wykonać na kilkunastu kolejnych stronach książki:

1. **Zainstaluj serwer bazy danych MySQL.**
2. **Zainstaluj sterownik bazy danych MySQL dla języka Python.**
3. **Utwórz bazę danych i tabele dla naszej aplikacji WWW.**
4. **Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.**

Po wykonaniu tych czterech zadań będziemy gotowi do wprowadzenia takich zmian w kodzie programu *vsearch4web.py*, aby logował on dane w bazie danych MySQL zamiast w pliku tekstowym. Następnie skorzystamy z języka SQL, aby zadać nasze pytania i — jeśli szczęście nam dopisze — uzyskać na nie odpowiedzi.

Nie istnieją głupie pytania

P: Czy musimy tu używać bazy MySQL?

O: Jeśli chcesz śledzić przykłady przedstawione w tym rozdziale, odpowiedź brzmi: tak.

P: Czy mogę korzystać z bazy MariaDB zamiast z MySQL-a?

O: Tak. Jako że MariaDB jest klonem MySQL-a, nie ma problemu, abyś używał tej bazy danych zamiast „oficjalnego" MySQL-a. (W rzeczywistości wśród devopsów pracujących w laboratorium *Rusz głową!* to właśnie MariaDB stanowi ulubioną bazę danych).

P: A co z bazą PostgreSQL? Mogę jej używać?

O: Hm... cóż, tak, ale z pewnym zastrzeżeniem: zamiast MySQL-a możesz spróbować zastosować bazę PostgreSQL, jeśli korzystasz już z niej (lub innego systemu zarządzania bazą danych wykorzystującego język SQL) i masz w tym pewne doświadczenie. Zwróć jednak uwagę na fakt, że w tym rozdziale nie pojawią się żadne instrukcje specyficzne dla bazy PostgreSQL (czy też jakiejś innej), możesz więc być zmuszony do samodzielnego wykonania pewnych eksperymentów, gdy coś, co będziemy prezentować za pomocą MySQL-a, nie będzie działać tak samo w przypadku wybranej przez Ciebie bazy danych. Do dyspozycji masz też zapewnianą przez Pythona niezależną, jednoużytkownikową bazę **SQLite**, która umożliwia korzystanie z języka SQL *bez* konieczności uruchamiania oddzielnego serwera. Ogólnie rzecz biorąc, wybór określonej technologii bazodanowej powinien być w dużej mierze uzależniony od tego, co próbujesz osiągnąć.

Zadanie nr 1: Zainstaluj serwer bazy danych MySQL

- [ ] Zainstaluj serwer bazy danych MySQL.
- [ ] Zainstaluj sterownik bazy danych MySQL dla języka Python.
- [ ] Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- [ ] Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Każde zakończone zadanie będziemy odhaczać na liście.

Jeśli masz już zainstalowaną bazę danych MySQL na swoim komputerze, możesz swobodnie przejść do zadania nr 2.

Przebieg procesu instalacji MySQL-a zależy od tego, z jakiego systemu operacyjnego korzystasz. Na szczęście ludzie odpowiedzialni za rozwój MySQL-a (i jego bliskiej kuzynki, bazy MariaDB) świetnie poradzili sobie z uproszczeniem tego procesu.

Jeśli korzystasz z systemu *Linux*, nie powinieneś mieć problemu ze znalezieniem pakietu `mysql-server` (lub `mariadb-server`) w swoim repozytorium oprogramowania. Użyj wybranego narzędzia instalacyjnego (`apt`, `aptitude`, `rpm`, `yum` czy co tam lubisz), aby zainstalować MySQL-a dokładnie w taki sposób, z jakiego skorzystałbyś w przypadku każdego innego pakietu.

Jeśli używasz systemu *Mac OS X*, polecamy Ci najpierw zainstalować *Homebrew* (informacje na jego temat znajdziesz pod adresem *http://brew.sh*), a następnie skorzystać z niego, aby zainstalować bazę MariaDB, ponieważ — jak wynika z naszych doświadczeń — ta kombinacja sprawdza się świetnie.

W przypadku wszystkich innych systemów operacyjnych (włączając wszystkie wersje systemu *Windows*) zalecamy Ci zainstalowanie wersji **Community Edition** serwera MySQL, która dostępna jest pod adresem:

https://dev.mysql.com/downloads/mysql/.

Albo jeśli wolisz korzystać z bazy MariaDB, przejdź pod adres:

https://mariadb.org/download/.

Koniecznie przeczytaj dokumentację instalacji związaną z wersją, którą zdecydujesz się pobrać i zainstalować.

Uwaga z działu marketingu: spośród wszystkich książek o MySQL-u... na całym świecie... to jest ta, którą przynieśliśmy do ~~baru~~... eee... znaczy: biura wtedy, gdy sami poznawaliśmy tę bazę danych.

Zapowiada się to bardzo boleśnie, ponieważ nigdy wcześniej nie korzystałam z MySQL-a...

Nie martw się, jeśli to dla Ciebie nowość.

Prezentując tu nasz materiał, nie oczekujemy, że staniesz się geniuszem od MySQL-a. Zapewnimy Ci wszystko, co jest potrzebne, aby dało się skutecznie uruchomić każdy z przedstawionych w tym rozdziale przykładów (nawet jeśli nigdy wcześniej nie miałeś do czynienia z MySQL-em).

Jeśli chcesz dowiedzieć się więcej o tym systemie bazy danych, polecamy Ci na początek lekturę doskonałej książki *SQL. Rusz głową!*, napisanej przez *Lynn Beighley* i wydanej przez wydawnictwo Helion.

Choć ta książka poświęcona jest językowi zapytań SQL, do prezentacji wszystkich przykładów wykorzystuje się w niej system zarządzania bazą danych MySQL. Mimo swojego wieku to nadal doskonałe źródło wiedzy na ten temat.

Wprowadzenie do DB-API języka Python

- [x] Zainstaluj serwer bazy danych MySQL.
- [ ] Zainstaluj sterownik bazy danych MySQL dla języka Python.
- [ ] Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- [ ] Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Gdy serwer bazy danych jest już zainstalowany, zostawmy go na chwilę w spokoju i dodajmy obsługę MySQL-a w Pythonie.

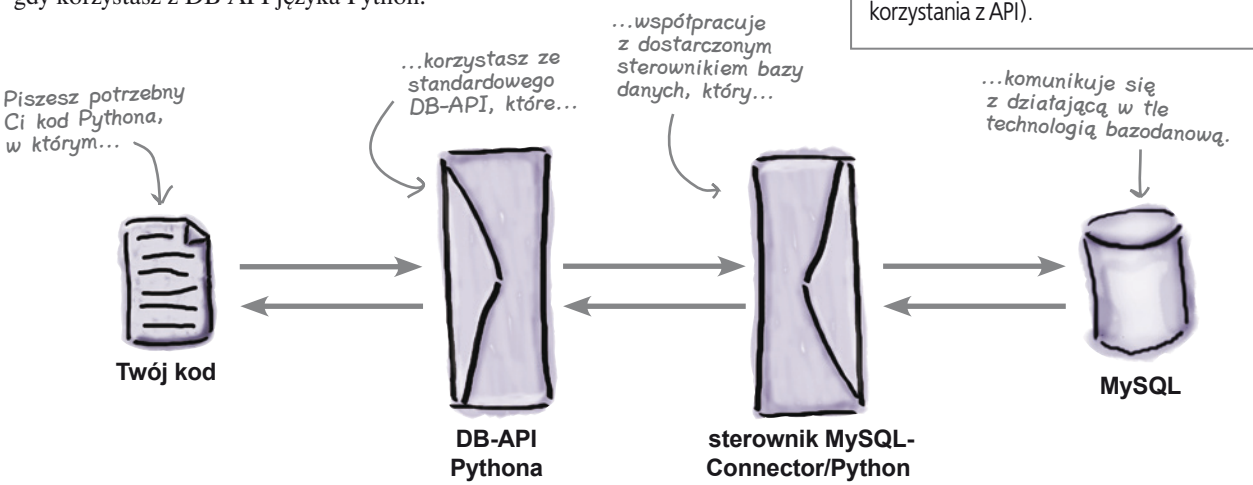
Interpreter Pythona jest standardowo wyposażony w pewne możliwości współpracy z bazami danych, ale nie ma wśród nich żadnego szczególnego wsparcia dla MySQL-a. Zapewniane jest tu standardowe bazodanowe API (ang. *application programmer interface*, czyli interfejs programisty aplikacji), znane pod nazwą *DB-API*, umożliwiające korzystanie z baz danych opartych na SQL-u. Tym, czego brakuje, jest **sterownik** (ang. *driver*) pozwalający podłączyć DB-API do systemu bazy danych, którego rzeczywiście używasz.

Zgodnie z konwencją, gdy programiści za pomocą Pythona wchodzą w interakcję z dowolną bazą danych, korzystają z DB-API niezależnie od tego, jaka technologia bazodanowa jest stosowana w określonej sytuacji. Jest to możliwe dzięki temu, że sterownik uwalnia ich od konieczności wgłębiania się w szczegóły techniczne dotyczące współpracy z faktycznym API konkretnej bazy danych, ponieważ DB-API zapewnia abstrakcyjną warstwę skutecznie rozdzielającą te dwa światy. Idea polega tu na tym, że korzystając z DB-API, możesz spokojnie wymienić używane w tle rozwiązanie bazodanowe, gdy okaże się to niezbędne, bez konieczności wyrzucania na śmietnik jakiegokolwiek istniejącego kodu.

Więcej do powiedzenia na temat DB-API będziemy mieć w dalszej części tego rozdziału. Poniżej znajdziesz graficzną reprezentację tego, co się dzieje, gdy korzystasz z DB-API języka Python.

Porady maniaka

DB-API Pythona zostało zdefiniowane w dokumencie PEB 249. Nie oznacza to bynajmniej, że musisz teraz rzucać wszystko i natychmiast zaczynać lekturę tego dokumentu. Został on opracowany przede wszystkim jako specyfikacja przydatna osobom implementującym sterowniki baz danych (nie ma więc być typowym materiałem szkoleniowym prezentującym sposoby korzystania z API).



Patrząc na powyższy schemat, niektórzy programiści stwierdzą zapewne, że korzystanie z DB-API Pythona musi być bardzo niewydajne. W końcu mamy tu *dwie* warstwy technologiczne występujące pomiędzy kodem a działającym w tle systemem bazy danych. Jednak korzystanie z DB-API umożliwia Ci zamianę działającej w tle bazy danych zgodnie z potrzebą, co z kolei pozwala uniknąć jakiegokolwiek „uwiązania" do konkretnego systemu bazodanowego, które zdarza się, gdy Twój kod odwołuje się *bezpośrednio* do bazy. Jeśli weźmiesz jeszcze pod uwagę, że nie ma dwóch dokładnie takich samych dialektów języka SQL, używanie DB-API okaże się prawdziwym wybawieniem, ponieważ zapewnia wyższy poziom abstrakcji.

Zadanie nr 2: Zainstaluj sterownik bazy danych MySQL dla języka Python

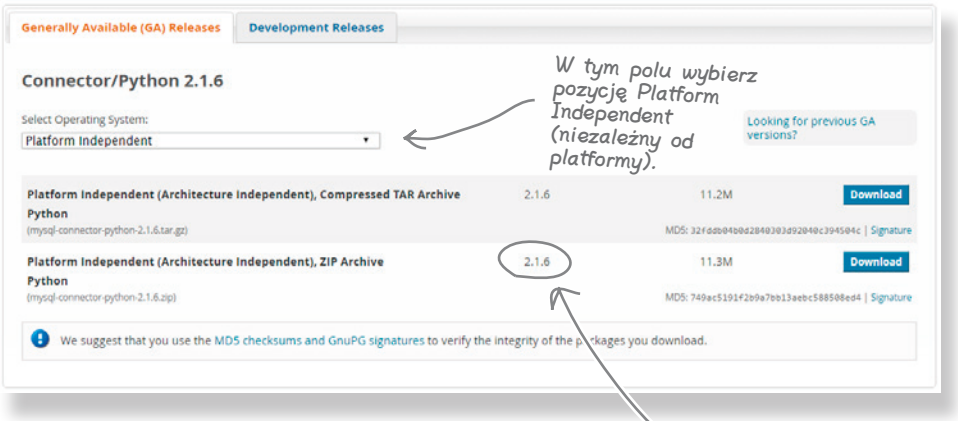
- [x] Zainstaluj serwer bazy danych MySQL.
- [ ] Zainstaluj sterownik bazy danych MySQL dla języka Python.
- [ ] Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- [ ] Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Każdy może samodzielnie napisać sterownik bazy danych (i wiele osób to robi), ale producenci baz dostarczają zwykle *oficjalne sterowniki* dla każdego z obsługiwanych przez nich języków programowania. Firma Oracle, producent technologii MySQL, zapewnia sterownik *MySQL-Connector/Python* i właśnie z niego mamy zamiar korzystać w tym rozdziale. Jest tu tylko jeden problem: sterownika tego nie da się zainstalować za pomocą narzędzia `pip`.

Czy oznacza to, że mamy pecha w kwestii możliwości zastosowania rozwiązania *MySQL-Connector/Python* w połączeniu z Pythonem? Nie, bynajmniej. Z faktu, że niezależny moduł nie korzysta z funkcji instalacyjnych narzędzia `pip`, rzadko wynikają jakieś poważniejsze kłopoty. Musimy po prostu zainstalować ten moduł „z palca". Wiąże się z tym pewna ilość dodatkowej pracy (w stosunku do tego, co byłoby w przypadku zastosowania narzędzia `pip`), ale nie jest to wielka ilość.

Zainstalujmy sterownik *MySQL-Connector/Python* ręcznie (mając na uwadze, że dostępne są również *inne* sterowniki, takie jak *PyMySQL*; mimo to wolimy zastosować rozwiązanie *MySQL-Connector/Python*, ponieważ jest to oficjalnie obsługiwany sterownik dostarczany przez twórców samego MySQL-a).

Zacznij od odwiedzenia strony pobierania sterownika *MySQL-Connector/Python* znajdującej się pod adresem *https://dev.mysql.com/downloads/connector/python/*. Po przejściu na nią prawdopodobnie zobaczysz, że z listy rozwijanej *Select Platform* (wybierz platformę) zostanie wstępnie wybrany system operacyjny, którego używasz. Zignoruj ten wybór i wskaż na liście pozycję *Platform Independent* (niezależny od platformy), tak jak zostało to pokazane poniżej.



W tym polu wybierz pozycję Platform Independent (niezależny od platformy).

Następnie kliknij jeden z widocznych przycisków *Download* (pobierz). W przypadku systemu *Windows* użytkownicy powinni zwykle pobrać plik ZIP, a użytkownicy systemów *Linux* oraz *Mac OS X* mogą skorzystać z piku GZ. Zapisz pobrany plik na swoim komputerze, a następnie kliknij go dwukrotnie, aby rozpakować tam, gdzie został zapisany.

Nie przejmuj się, jeśli numer wersji, który widzisz na stronie pobierania, jest inny niż ten przedstawiony powyżej. Dopóki jest to numer najnowszej wersji rozwiązania, wszystko jest w jak najlepszym porządku.

Zainstaluj sterownik MySQL-Connector/Python

- [x] Zainstaluj serwer bazy danych MySQL.
- [ ] Zainstaluj sterownik bazy danych MySQL dla języka Python.
- [ ] Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- [ ] Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Jeśli pobrałeś już sterownik i rozpakowałeś go na swoim komputerze, otwórz okno terminala w nowo utworzonym folderze (jeśli korzystasz z systemu *Windows*, otwórz terminal za pomocą polecenia *Uruchom jako administrator*).

Folder ten nosi w przypadku naszego komputera nazwę *mysql-connector-python-2.1.6* i został utworzony w folderze *Pobrane*. Aby zainstalować sterownik w systemie *Windows*, wydaj przedstawione poniżej polecenie we wspomnianym przed chwilą folderze zawierającym pliki sterownika.

```
py -3 setup.py install
```

W systemie *Linux* lub *Mac OS X* powinieneś zamiast niego skorzystać z polecenia pokazanego poniżej.

```
sudo -H python3 setup.py install
```

Niezależnie od tego, którego systemu operacyjnego używasz, wydanie odpowiedniego spośród zaprezentowanych poleceń spowoduje wyświetlenie na ekranie serii komunikatów, które powinny wyglądać jak te przedstawione poniżej.

```
running install
Not Installing C Extension
running build
running build_py
creating build
creating build\lib
creating build\lib\mysql
copying lib\mysql\__init__.py -> build\lib\mysql
creating build\lib\mysql\connector
...
running install_egg_info
Writing C:\Users\lsuma\AppData\Local\Programs\Python\Python36\Lib\site-packages\mysql_
connector_python-2.1.6-py3.6.egg-info
```

Pokazane tu ścieżki dostępu mogą być inne na Twoim komputerze. Nie martw się, jeśli tak właśnie jest.

Jeśli instalujesz moduł za pomocą narzędzia `pip`, przeprowadza ono ten sam proces, ale ukrywa przed Tobą powyższe komunikaty. To, co tu widać, to komunikaty stanu wskazujące, że instalacja przebiega gładko. Gdy coś idzie źle, na ekranie powinien się pojawić komunikat błędu zawierający wystarczająco dużo informacji, abyś mógł rozwiązać występujący problem. Jeśli wszystko jest w porządku, obecność powyższych komunikatów potwierdza, że sterownik *MySQL-Connector/Python* jest gotów do użycia.

Nie istnieją głupie pytania

P: Czy powinienem się martwić komunikatem „Not Installing C Extension" (nie instaluje rozszerzenia C)?

O: Nie. Moduły pochodzące od dostawców niezależnych czasami zawierają osadzony kod C, który może poprawiać wydajność przetwarzania wymagającego obliczeniowo. Jednak nie wszystkie systemy operacyjne zapewniają preinstalowany kompilator C, dlatego musisz wyraźnie poprosić o włączenie obsługi rozszerzenia, gdy moduł jest instalowany (jeśli stwierdzisz, że jest Ci to potrzebne). Jeśli tego nie zrobisz, mechanizm instalacji niezależnego modułu skorzysta z (potencjalnie wolniejszego) kodu Pythona zamiast kodu C. Dzięki temu moduł może działać na dowolnej platformie, niezależnie od obecności kompilatora C. Jeśli niezależny moduł używa *wyłącznie* kodu Pythona, mówi się, że jest napisany w „czystym Pythonie". W przykładzie przedstawionym powyżej zainstalowaliśmy czysto pythonową wersję sterownika *MySQL-Connector/Python*.

Zadanie nr 3: Utwórz bazę danych i tabele dla naszej aplikacji WWW

- [x] Zainstaluj serwer bazy danych MySQL.
- [x] Zainstaluj sterownik bazy danych MySQL dla języka Python.
- [ ] Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- [ ] Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Masz już zainstalowany na swoim komputerze serwer bazy danych MySQL oraz sterownik *MySQL-Connector/Python*. Czas na zadanie nr 3, które wymaga utworzenia bazy danych i tabel potrzebnych naszej aplikacji.

Aby to zrobić, powinieneś skomunikować się z serwerem MySQL, używając zapewnianego przezeń narzędzia, które uruchamia się w systemowym wierszu poleceń i z jego poziomu wykorzystuje. Narzędzie to znane jest jako *konsola* MySQL. Poniżej znajdziesz polecenie, za pomocą którego możesz ją uruchomić[1], logując się jako administrator bazy danych MySQL (używający identyfikatora użytkownika `root`).

```
mysql  -u root -p
```

Jeśli w czasie instalacji serwera MySQL ustawiłeś hasło administratora, wpisz je po naciśnięciu klawisza *Enter*. Jeśli nie ustawiłeś tego hasła, po prostu naciśnij klawisz *Enter* dwa razy. Zostaniesz w wyniku tego przeniesiony do **wiersza poleceń konsoli** i zobaczysz jej znak zachęty, który wygląda tak, jak zostało to pokazane poniżej (po lewej — dla serwera MySQL, po prawej — dla serwera MariaDB).

```
mysql>                                  MariaDB [None]>
```

Wszystkie polecenia, które wprowadzisz po znaku zachęty konsoli, są przekazywane serwerowi MySQL w celu wykonania. Zacznijmy od utworzenia bazy danych dla naszej aplikacji WWW. Przypomnij sobie, że chcemy korzystać z bazy danych w celu przechowywania logowanych danych, dlatego jej nazwa powinna odzwierciedlać jej przeznaczenie. Nazwijmy ją zatem `vsearchlogDB`. Poniżej znajdziesz polecenie konsoli, za pomocą którego można utworzyć tę bazę danych.

```
mysql> create database vsearchlogDB;
```

Pamiętaj, aby każde polecenie, które wprowadzasz w konsoli MySQL, kończyć znakiem średnika.

Konsola zareaguje na wydanie tego polecenia (dość tajemniczym) komunikatem stanu: `Query OK, 1 row affected (0.01 sec)`, który oznacza, że zapytanie było w porządku i że miało wpływ na jeden wiersz. To używany przez konsolę sposób powiadamiania, że wszystko poszło dobrze.

Utwórzmy identyfikator i hasło użytkownika bazy danych przeznaczone specjalnie dla naszej aplikacji WWW. Będą one przez nią używane podczas komunikacji z MySQL-em zamiast identyfikatora użytkownika `root` (to ostatnie jest uważane za złą praktykę). Kolejne przedstawione polecenie umożliwia utworzenie nowego użytkownika bazy, o nazwie `vsearch`, korzystającego z hasła `vsearchpasswd`, oraz nadanie mu pełnych uprawnień do bazy danych `vsearchlogDB`.

```
mysql> grant all on vsearchlogDB.* to 'vsearch' identified by 'vsearchpasswd';
```

Możesz tu użyć innego hasła, jeśli tylko chcesz. Pamiętaj jedynie, aby w przykładzie, który wkrótce zaprezentujemy, korzystać z własnego hasła, a nie naszego.

Na ekranie pojawi się komunikat stanu `Query OK` podobny do wyświetlonego poprzednim razem. Będzie on stanowił potwierdzenie, że użytkownik został utworzony. Teraz wylogujmy się z konsoli za pomocą polecenia widocznego poniżej.

```
mysql> quit
```

W wyniku tego zobaczysz przyjazny komunikat pożegnalny `Bye`, a następnie powrócisz do wiersza poleceń swojego systemu operacyjnego.

[1] W razie problemu z uruchomieniem konsoli (związanym z nierozpoznaniem polecenia przez system operacyjny) należy przejść do miejsca, w którym dostępny jest program *mysql.exe* (lub ustawić odpowiednią ścieżkę dostępu w zmiennych systemowych). W systemie *Windows* program ten znajduje się w folderze *C:\Program Files\MySQL\MySQL Server 5.7\bin\ — przyp. tłum.*

Wybierz strukturę dla swojego logu

- [x] Zainstaluj serwer bazy danych MySQL.
- [x] Zainstaluj sterownik bazy danych MySQL dla języka Python.
- [ ] Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- [ ] Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Mając już gotową bazę danych dla swojej aplikacji WWW, możesz utworzyć dowolną liczbę tabel w tej bazie (zgodnie z wymaganiami Twojej aplikacji, oczywiście). W naszym przypadku wystarczy nam pojedyncza tabela, ponieważ musimy przechowywać jedynie dane związane z poszczególnymi zalogowanymi żądaniami sieciowymi.

Przypomnij sobie sposób, w jaki w poprzednim rozdziale przechowywaliśmy te dane w pliku tekstowym. Każdy wiersz pliku *vsearch.log* zawierał dane dla jednego żądania i miał określony format, który został zaprezentowany poniżej.

Logujemy wartość argumentu phrase...

...jak również wartość argumentu letters.

Logowany jest także adres IP komputera, który przesłał dane.

```
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'t', 'y'}
```

Mamy tu (raczej długi) łańcuch znakowy, który opisuje używaną przeglądarkę internetową.

Jako ostatni — choć nie najmniej ważny — logowany jest faktyczny wynik otrzymany w wyniku wyszukiwania znaków znajdujących się w argumencie letters w łańcuchu przechowywanym w argumencie phrase.

Tabela, której potrzebujesz, powinna mieć co najmniej pięć pól: dla frazy, liter, adresu IP, łańcucha znakowego przeglądarki oraz wartości wyników. Dodajmy jednak jeszcze dwa inne pola: unikatowy identyfikator każdego zalogowanego żądania oraz znacznik czasu (ang. *timestamp*) rejestrujący, kiedy żądanie zostało zalogowane. Jako że te dwa ostatnie pola występują niezwykle często, MySQL zapewnia prosty sposób dołączania tych danych do każdego zalogowanego żądania, co pokazano w dolnej części tej strony.

Za pomocą konsoli możesz określić strukturę tabeli, którą zamierzasz utworzyć. Zanim to jednak zrobisz, zaloguj się jako nasz nowo utworzony użytkownik `vsearch`, korzystając z przedstawionego polecenia (i podając prawidłowe hasło po naciśnięciu klawisza *Enter*).

```
mysql -u vsearch -p vsearchlogDB
```

Pamiętaj, że dla tego użytkownika ustawiliśmy hasło „vsearchpasswd".

Poniżej znajdziesz instrukcję SQL, z której skorzystaliśmy, aby utworzyć wymaganą tabelę (o nazwie `log`). Zauważ, że symbol `->` nie stanowi części tej instrukcji, lecz jest automatycznie dodawany przez konsolę w celu wskazania, że spodziewa się ona od Ciebie wprowadzenia dalszego ciągu kodu (gdy Twój kod SQL rozciąga się na wiele wierszy). Instrukcja się kończy (i zostaje wykonana), gdy wpiszesz znak średnika i naciśniesz klawisz *Enter*.

```
mysql> create table log (
    -> id int auto_increment primary key,
    -> ts timestamp default current_timestamp,
    -> phrase varchar(128) not null,
    -> letters varchar(32) not null,
    -> ip varchar(16) not null,
    -> browser_string varchar(256) not null,
    -> results varchar(64) not null );
```

MySQL automatycznie zapewni dane dla tych pól.

To jest symbol kontynuacji stosowany przez konsolę.

Te pola będą przechowywać dane dla każdego żądania (w takiej postaci, w jakiej są wprowadzane w formularzu).

Potwierdź, że Twoja tabela jest gotowa na przyjęcie danych

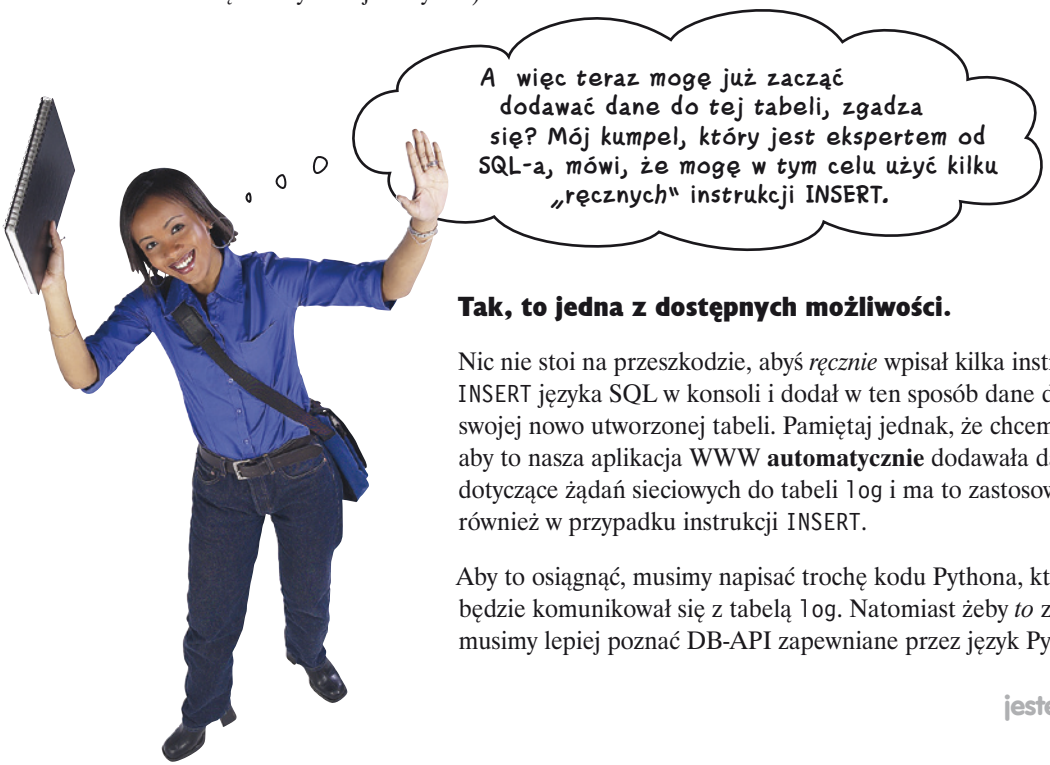
- [x] Zainstaluj serwer bazy danych MySQL.
- [x] Zainstaluj sterownik bazy danych MySQL dla języka Python.
- [x] Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- [ ] Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Wraz z utworzeniem tabeli zakończyliśmy zadanie nr 3.

Przekonajmy się za pomocą konsoli, że tabela rzeczywiście powstała i że ma strukturę, którą zdefiniowaliśmy. Będąc nadal zalogowanym w konsoli MySQL jako użytkownik `vsearch`, wydaj po jej znaku zachęty polecenie ***describe log***.

```
mysql> describe log;
+----------------+--------------+------+-----+-------------------+----------------+
| Field          | Type         | Null | Key | Default           | Extra          |
+----------------+--------------+------+-----+-------------------+----------------+
| id             | int(11)      | NO   | PRI | NULL              | auto_increment |
| ts             | timestamp    | NO   |     | CURRENT_TIMESTAMP |                |
| phrase         | varchar(128) | NO   |     | NULL              |                |
| letters        | varchar(32)  | NO   |     | NULL              |                |
| ip             | varchar(16)  | NO   |     | NULL              |                |
| browser_string | varchar(256) | NO   |     | NULL              |                |
| results        | varchar(64)  | NO   |     | NULL              |                |
+----------------+--------------+------+-----+-------------------+----------------+
```

I oto mamy dowód, że tabela `log` istnieje i ma strukturę, która odpowiada wymaganiom naszej aplikacji WWW związanym z logowaniem. Wprowadź polecenie ***quit***, aby opuścić konsolę (ponieważ na razie nie będziemy z niej korzystać).



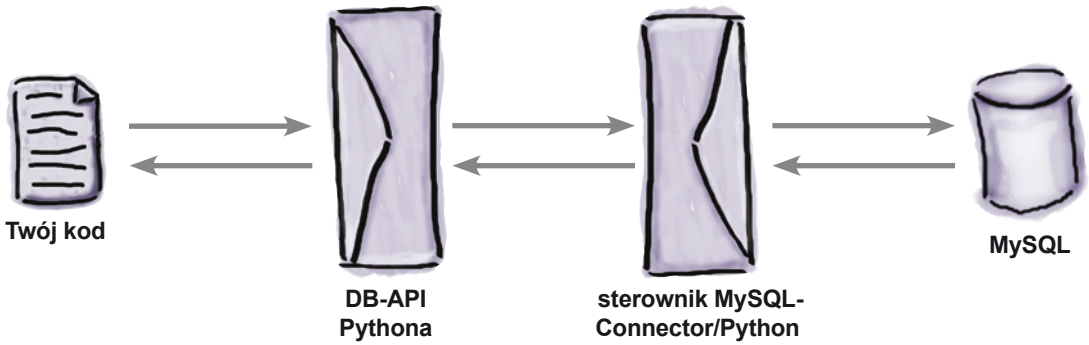
Tak, to jedna z dostępnych możliwości.

Nic nie stoi na przeszkodzie, abyś *ręcznie* wpisał kilka instrukcji `INSERT` języka SQL w konsoli i dodał w ten sposób dane do swojej nowo utworzonej tabeli. Pamiętaj jednak, że chcemy, aby to nasza aplikacja WWW **automatycznie** dodawała dane dotyczące żądań sieciowych do tabeli `log` i ma to zastosowanie również w przypadku instrukcji `INSERT`.

Aby to osiągnąć, musimy napisać trochę kodu Pythona, który będzie komunikował się z tabelą `log`. Natomiast żeby *to* zrobić, musimy lepiej poznać DB-API zapewniane przez język Python.

DB-API z bliska (1. z 3)

Przypomnij sobie schemat przedstawiony we wcześniejszej części tego rozdziału, na którym pokazano miejsce zajmowane przez DB-API Pythona w odniesieniu do Twojego kodu, wybranego przez Ciebie sterownika bazy danych oraz działającego w tle systemu bazy danych.



Zastosowanie DB-API ma Ci umożliwić wymianę używanej kombinacji sterownika i bazy danych przy bardzo niewielkich zmianach w Twoim kodzie Pythona, jeśli tylko ograniczysz się do korzystania wyłącznie z możliwości zapewnianych przez to API.

Dokonajmy przeglądu tego, z czym wiąże się programowanie w zgodzie z tym ważnym standardem Pythona. Zamierzamy tu przedstawić sześć kroków.

DB-API — krok 1. Zdefiniuj właściwości swojego połączenia

Łącząc się z bazą danych MySQL, potrzebujesz czterech informacji: (1) adresu IP lub nazwy komputera, na którym działa serwer MySQL (znanego także jako *host*), (2) identyfikatora użytkownika, który ma być wykorzystywany, (3) hasła związanego z tym identyfikatorem użytkownika oraz (4) nazwy bazy danych, z którą użytkownik o wskazanym identyfikatorze chce współpracować.

Sterownik *MySQL-Connector/Python* umożliwia umieszczenie tych właściwości połączenia w słowniku Pythona, dzięki czemu łatwo jest ich używać i odwoływać się do nich. Zróbmy to teraz, wprowadzając kod w wierszu poleceń powłoki Pythona. Wykonuj na swoim komputerze te same operacje co my. Poniżej przedstawiona została definicja słownika (o nazwie `dbconfig`), który łączy cztery wymagane „klucze połączenia” z odpowiadającymi im wartościami.

1. Nasz serwer działa na komputerze lokalnym, dlatego w przypadku klucza 'host' korzystamy z IP localhosta.

```
>>> dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }
```

2. Identyfikator użytkownika vsearch przedstawiony wcześniej w tym rozdziale został przypisany do klucza 'user'.

3. Kluczowi 'password' jest przypisane prawidłowe hasło, które ma być stosowane dla tego identyfikatora użytkownika.

4. Nazwa bazy danych — w tym przypadku „vsearchlogDB” — jest przypisana do klucza 'database'.

DB-API — krok 2. Zaimportuj sterownik swojej bazy danych

Gdy masz już zdefiniowane właściwości połączenia, nadszedł czas na zaimportowanie sterownika bazy danych.

```
>>> import mysql.connector
```

Zaimportuj sterownik bazy danych, której będziesz używał.

Operacja ta sprawia, że sterownik specyficzny dla bazy MySQL staje się dostępny dla DB-API.

DB-API — krok 3. Nawiąż połączenie z serwerem

Nawiążmy połączenie z naszym serwerem, korzystając z funkcji `connect` zapewnianej przez DB-API. Referencję do tego połączenia zapiszmy w zmiennej o nazwie `conn`. Poniżej znajdziesz wywołanie funkcji `connect`, dzięki któremu program nawiązuje połączenie z serwerem bazy danych MySQL (a także tworzy zmienną `conn`).

```
>>> conn = mysql.connector.connect(**dbconfig)
```

To wywołanie powoduje nawiązanie połączenia.

Przekaż jako argument słownik zawierający właściwości połączenia.

Zauważ dziwne dwa znaki gwiazdki (**) poprzedzające pojedynczy argument wywołania funkcji `connect`. (Jeśli jesteś programistą C lub C++, **nie** interpretuj tego zapisu jako „wskaźnika do wskaźnika", ponieważ w Pythonie w ogóle nie funkcjonuje pojęcie wskaźników). Notacja ** informuje funkcję `connect`, że za pomocą pojedynczej zmiennej (którą w naszym przypadku jest utworzony przez Ciebie słownik `dbconfig`) przekazywany jest słownik argumentów. Napotykając zapis **, funkcja `connect` rozwija pojedynczy argument słownikowy do postaci czterech oddzielnych argumentów, które są następnie przez nią wykorzystywane do nawiązania połączenia. (Więcej przykładów zastosowania tej notacji znajdziesz w dalszych rozdziałach, na razie po prostu używaj jej w zaprezentowany tu sposób).

DB-API — krok 4. Otwórz kursor

Aby wysyłać polecenia SQL do Twojej bazy danych (za pośrednictwem otwartego właśnie połączenia), jak również aby otrzymywać od niej wyniki, potrzebujesz *kursora*. O kursorze możesz myśleć jako o bazodanowym odpowiedniku *uchwytu pliku*, o którym była mowa w poprzednim rozdziale niniejszej książki (i który umożliwia Ci komunikowanie się z plikiem dyskowym, gdy zostanie on już otwarty).

```
>>> cursor = conn.cursor()
```

Utwórz kursor, aby móc wysyłać polecenia do serwera i otrzymywać od niego wyniki.

Tworzenie kursora jest bardzo proste. Robi się to, wywołując metodę `cursor` udostępnianą przez każdy obiekt połączenia. Podobnie jak ma to miejsce w przedstawionym powyżej przypadku połączenia, tu również zapisuje się w zmiennej referencję do tworzonego kursora (którą w nagłym przypływie dzikiej kreatywności nazwiemy `cursor`).

Jesteśmy teraz gotowi do wysyłania poleceń SQL do serwera oraz — mamy nadzieję — otrzymywać od niego wyniki.

Zanim jednak zaczniemy to robić, poświęćmy chwilę na przegląd kroków wykonanych do tej pory. Zdefiniowaliśmy właściwości połączenia z bazą danych, zaimportowaliśmy moduł sterownika, utworzyliśmy obiekt połączenia oraz kursor. Niezależnie od bazy danych, z której korzystasz, kroki te są wymagane w przypadku wszystkich interakcji z MySQL-em (zmieniają się tu tylko właściwości połączenia). Pamiętaj o tym, gdy będziesz korzystał ze swoich danych za pośrednictwem kursora.

DB-API z bliska (2. z 3)

Po utworzeniu kursora i przypisaniu go do zmiennej pora na rozpoczęcie interakcji z danymi przechowywanymi w Twojej bazie danych. Dostęp do danych uzyskasz za pomocą języka zapytań SQL.

DB-API — krok 5. Weź się za SQL-a!

Zmienna `cursor` umożliwia Ci wysyłanie zapytań SQL do bazy MySQL, jak również odbieranie od niej wszystkich wyników przetwarzania zapytań.

Ogólną zasadą stosowaną przez programistów Pythona pracujących w laboratorium *Rusz głową!* jest zapisywanie kodu SQL, który ma zostać przesłany do serwera bazy danych, w potrójnych znakach cudzysłowu, a następnie przypisywanie tak utworzonego łańcucha znakowego do zmiennej o nazwie `_SQL`. Łańcuch tego rodzaju jest w tym przypadku stosowany dlatego, że zapytania SQL często rozciągają się na wiele wierszy, a użycie potrójnych znaków cudzysłowu tymczasowo wyłącza używaną przez interpreter Pythona regułę stanowiącą, że koniec wiersza oznacza koniec instrukcji. Korzystanie z nazwy zmiennej `_SQL` jest konwencją przyjętą przez programistów pracujących w laboratorium *Rusz głową!* w przypadku definiowania stałych wartości w Pythonie, możesz jednak używać tu dowolnej nazwy zmiennej (i bynajmniej nie musi ona być zapisywana wyłącznie wielkimi literami ani poprzedzana znakiem podkreślenia).

Zacznijmy od zapytania MySQL-a o nazwy tabel należących do bazy danych, z którą mamy połączenie. Aby to zrobić, przypisz zapytanie `show tables` do zmiennej `_SQL`, a następnie wywołaj metodę `cursor.execute`, przekazując tę zmienną jako argument.

Przypisz zapytanie SQL do zmiennej.

```
>>> _SQL = """show tables"""
>>> cursor.execute(_SQL)
```

Wyślij zapytanie przechowywane w zmiennej _SQL do bazy MySQL w celu wykonania.

Gdy wpiszesz pokazane powyżej wywołanie metody `cursor.execute` po znaku zachęty `>>>` i naciśniesz klawisz *Enter*, Twoje zapytanie SQL zostanie wysłane do serwera MySQL, który rozpocznie jego wykonywanie (oczywiście, jeśli tylko jest ono prawidłowe i zgodne z językiem SQL). Na ekranie *nie* pojawią się jednak natychmiast jakiekolwiek wyniki tego zapytania; musisz o nie poprosić.

Możesz to zrobić za pomocą jednej z trzech metod:

- `cursor.fetchone` umożliwia otrzymanie **jednego** wiersza wyników;
- `cursor.fetchmany` umożliwia otrzymanie takiej **liczby** wierszy, jaką określisz;
- `cursor.fetchall` umożliwia otrzymanie **wszystkich** wierszy tworzących wyniki.

Teraz skorzystajmy z metody `cursor.fetchall`, aby pobrać wszystkie wyniki wygenerowane wskutek wykonania przedstawionego powyżej zapytania, i przypiszmy te wyniki do zmiennej o nazwie `res`, a następnie wyświetlmy zawartość tej zmiennej w wierszu poleceń powłoki Pythona.

Pobierz wszystkie dane zwrócone przez bazę MySQL.

```
>>> res = cursor.fetchall()
>>> res
[('log',)]
```

Wyświetl uzyskane wyniki.

Zawartość zmiennej `res` wygląda nieco dziwnie, nie sądzisz? Prawdopodobnie spodziewałeś się tu zobaczyć pojedyncze słowo, ponieważ z wcześniejszej części tego rozdziału wiesz, że nasza baza danych (`vsearchlogDB`) zawiera jedną tabelę o nazwie `log`. Metoda `cursor.fetchall` zwraca jednak zawsze *listę krotek*, nawet wtedy, gdy zwracany jest tylko jeden element danych (jak dzieje się to w naszym przypadku). Przyjrzyjmy się innemu przykładowi — takiemu, w którym baza MySQL zwraca nieco więcej danych.

Nasze kolejne zapytanie, `describe log`, umożliwia uzyskanie informacji na temat tabeli `log`, która jest przechowywana w bazie danych. Jak możesz przekonać się poniżej, informacje te zostały wyświetlone *dwukrotnie*: raz w swojej surowej formie (która jest nieco chaotyczna), a następnie ponownie w wielu wierszach. Pamiętaj, że wyniki zwracane przez metodę `cursor.fetchall` są listą krotek.

Poniżej znajdziesz wyniki kolejnego wywołania metody `cursor.fetchall`.

```
>>> _SQL = """describe log"""
>>> cursor.execute(_SQL)
>>> res = cursor.fetchall()
>>> res
[('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment'), ('ts', 'timestamp',
'NO', '', 'CURRENT_TIMESTAMP', ''), ('phrase', 'varchar(128)', 'NO', '', None,
''), ('letters', 'varchar(32)', 'NO', '', None, ''), ('ip', 'varchar(16)',
'NO', '', None, ''), ('browser_string', 'varchar(256)', 'NO', '', None, ''),
('results', 'varchar(64)', 'NO', '', None, '')]
```

Weź zapytanie SQL...

...następnie wyślij je do serwera...

...a potem odwołaj się do wyników.

Wygląda to trochę chaotycznie, ale jest to lista krotek.

```
>>> for row in res:
      print(row)
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('ts', 'timestamp', 'NO', '', 'CURRENT_TIMESTAMP', '')
('phrase', 'varchar(128)', 'NO', '', None, '')
('letters', 'varchar(32)', 'NO', '', None, '')
('ip', 'varchar(16)', 'NO', '', None, '')
('browser_string', 'varchar(256)', 'NO', '', None, '')
('results', 'varchar(64)', 'NO', '', None, '')
```

Weź każdy wiersz należący do wyników...

...i wyświetl go w osobnym wierszu.

Każda krotka należąca do listy zajmuje teraz osobny wiersz.

Efekt wyświetlania danych wiersz po wierszu, który został pokazany powyżej, nie jest może o niebo lepszy niż zaprezentowane wcześniej surowe dane, ale porównaj go z danymi wyjściowymi pokazanymi przez konsolę MySQL kilka stron wcześniej (i zamieszczonymi poniżej). To, co widać powyżej, odpowiada temu, co możesz zobaczyć poniżej, różnica polega tylko na tym, że teraz dane te umieszczone są w strukturze danych języka Python o nazwie `res`.

```
mysql> describe log;
+----------------+--------------+------+-----+-------------------+----------------+
| Field          | Type         | Null | Key | Default           | Extra          |
+----------------+--------------+------+-----+-------------------+----------------+
| id             | int(11)      | NO   | PRI | NULL              | auto_increment |
| ts             | timestamp    | NO   |     | CURRENT_TIMESTAMP |                |
| phrase         | varchar(128) | NO   |     | NULL              |                |
| letters        | varchar(32)  | NO   |     | NULL              |                |
| ip             | varchar(16)  | NO   |     | NULL              |                |
| browser_string | varchar(256) | NO   |     | NULL              |                |
| results        | varchar(64)  | NO   |     | NULL              |                |
+----------------+--------------+------+-----+-------------------+----------------+
```

Przyjrzyj się uważnie. To te same dane.

DB-API z bliska (3. z 3)

Użyjmy zapytania `insert`, aby dodać trochę przykładowych danych do tabeli `log`.

Bardzo kusząca wydaje się perspektywa przypisania przedstawionego poniżej zapytania (które napisaliśmy w wielu wierszach) do zmiennej `_SQL` i wywołania metody `cursor.execute` w celu wysłania go do serwera.

```
>>> _SQL = """insert into log
               (phrase, letters, ip, browser_string, results)
               values
               ('mierni', 'aeiou', '127.0.0.1', 'Firefox', "{'e', 'i'}")"""
>>> cursor.execute(_SQL)
```

Nie zrozum nas źle, z zapytaniem tym jest wszystko w porządku i zadziałałoby ono bez kłopotu. Jednak *zapisywanie na sztywno* wartości danych w ten sposób raczej nie będzie tym, co chciałbyś robić, ponieważ najpewniej będą się one zmieniać wraz z każdym kolejnym zapytaniem `insert`, za pomocą którego będziesz umieszczał je w swojej tabeli. Pamiętaj, że w tabeli `log` planujesz logować szczegóły związane z każdym zapytaniem sieciowym, co oznacza, że wartości danych *będą* się zmieniać wraz z kolejnymi żądaniami, dlatego zapisywanie ich na sztywno w sposób przedstawiony powyżej byłoby prawdziwą katastrofą.

Aby pozwolić Ci uniknąć konieczności korzystania z zapisanych na sztywno danych (z jakimi mamy do czynienia powyżej), DB-API zapewniane przez Pythona umożliwia umieszczanie „symboli zastępczych danych" w łańcuchach znakowych zapytań. Symbole te są zastępowane rzeczywistymi wartościami w momencie wywołania metody `cursor.execute`. Dzięki temu możesz wielokrotnie używać tego samego zapytania z wieloma różnymi wartościami danych, przekazując je jako argumenty zapytania tuż przed tym, zanim zostanie ono wykonane. Symbole zastępcze w zapytaniu to wartości tekstowe i są identyfikowane w kodzie za pomocą ciągu `%s`, tak jak pokazano to poniżej.

Porównaj poniższe polecenia z tymi przedstawionymi powyżej.

Komponując swoje zapytanie, skorzystaj z symboli zastępczych zapewnianych przez DB-API zamiast z rzeczywistych wartości danych.

```
>>> _SQL = """insert into log
               (phrase, letters, ip, browser_string, results)
               values
                (%s, %s, %s, %s, %s)"""
>>> cursor.execute(_SQL, ('mierni', 'xyz', '127.0.0.1', 'Safari', 'set()'))
```

Powyżej znajdują się dwie rzeczy warte odnotowania. Po pierwsze, zamiast zapisywać na sztywno rzeczywiste wartości danych bezpośrednio w zapytaniu SQL, użyliśmy symboli zastępczych `%s` informujących DB-API, że ma się ono spodziewać wartości tekstowych, które zastąpią te symbole przed wykonaniem zapytania. Jak widać, powyżej jest pięć tych symboli zastępczych `%s`, dlatego drugą rzeczą do odnotowania jest to, że wywołanie metody `cursor.execute` będzie się spodziewać pięciu dodatkowych parametrów. Jedyny problem jest tutaj taki, że metoda ta nie przyjmuje *dowolnej* liczby parametrów; spodziewa się ich *najwyżej* dwóch.

Jak to możliwe?

Gdy spojrzysz na ostatni wiersz kodu, od razu stanie się dla Ciebie jasne, że metoda `cursor.execute` przyjmuje tych *pięć* wartości danych, które zostały jej przekazane (i nie narzeka), co jest więc grane?

Przyjrzyj się raz jeszcze temu wierszowi kodu, tym razem nieco uważniej. Czy widzisz nawias obejmujący nasze wartości danych? Zastosowanie go powoduje, że te pięć elementów zostaje przekształconych w pojedynczą krotkę (zawierającą indywidualne dane). Dzięki temu w ostatnim wierszu powyższego kodu metoda `cursor.execute` otrzymuje tak naprawdę tylko dwa argumenty: zapytanie zawierające symbole zastępcze oraz pojedynczą krotkę zawierającą odpowiednie wartości danych.

A zatem gdy kod przedstawiony na tej stronie jest wykonywany, wartości danych są umieszczane w tabeli `log`, zgadza się? Cóż... niezupełnie.

Gdy korzystasz z metody `cursor.execute` w celu wysłania danych do systemu bazodanowego (za pomocą zapytania `insert`), dane wcale nie muszą zostać natychmiast zapisane w bazie. Dzieje się tak, ponieważ zapis w bazie danych jest **kosztowną** operacją (z punktu widzenia wydajności przetwarzania), dlatego wiele systemów bazodanowych zbiera instrukcje `insert` w pamięci podręcznej (ang. *cache*), a następnie wykonuje je wszystkie naraz w późniejszym czasie. Może to czasami oznaczać, że danych, co do których masz pewność, że znajdują się już w bazie danych, w rzeczywistości *jeszcze* w niej nie ma, a to może prowadzić do wystąpienia pewnych problemów.

Jeśli na przykład skorzystasz z zapytania `insert`, aby wysłać dane do tabeli, a następnie natychmiast użyjesz zapytania `select`, aby je odczytać, dane mogą nie być dostępne, ponieważ nadal przebywają w pamięci podręcznej systemu bazodanowego, oczekując na zapisanie. Jeśli tak się stanie, masz pecha, jako że zapytanie `select` nie zwróci Ci żadnych danych. Ostatecznie dane zostaną zapisane, a więc nie przepadną, ale to standardowe zachowanie mechanizmu pamięci podręcznej może Cię nie satysfakcjonować.

Jeśli pogorszenie wydajności związane z zapisem danych w bazie nie stanowi dla Ciebie problemu, możesz zmusić system bazodanowy do zachowania w tabeli wszystkich potencjalnie oczekujących danych za pomocą metody `conn.commit`. Zróbmy to teraz, aby mieć pewność, że dwie instrukcje `insert` zaprezentowane na poprzedniej stronie zostały wykonane w stosunku do tabeli `log`. Gdy Twoje dane zostaną już zapisane, będziesz mógł użyć zapytania `select`, aby potwierdzić, że rzeczywiście znajdują się one w bazie danych, co pokazano poniżej.

Wymuś" zapis w tabeli wszystkich danych przechowywanych w pamięci podręcznej.

```
>>> conn.commit()
>>> _SQL = """select * from log"""
>>> cursor.execute(_SQL)
>>> for row in cursor.fetchall():
    print(row)

(1, datetime.datetime(2017, 6, ..., "{'e', 'i'}")
(2, datetime.datetime(2017, 6, ..., 'set()')
```

Pobierz zapisane przed chwilą dane.

Tu widać wartość identyfikatora (id) przypisaną automatycznie do tego wiersza przez system MySQL...

Skróciliśmy nieco dane wyjściowe, aby mogły się zmieścić na tej stronie.

...a tutaj jest wartość, którą przypisał on znacznikowi czasu (ts).

Powyżej możesz zobaczyć, że podczas wstawiania danych do wiersza MySQL automatycznie określił prawidłowe wartości, które powinny się pojawić w polach `id` oraz `ts`. Dane zwrócone przez serwer bazy danych stanowią (podobnie jak miało to miejsce wcześniej) listę krotek. Zamiast zapisywać wyniki wywołania metody `cursor.fetchall` w zmiennej, która jest następnie używana w pętli, używamy jej tam bezpośrednio. Nie zapominaj również, że krotka to niemodyfikowalna lista i jako taka umożliwia korzystanie ze zwykłej notacji nawiasów kwadratowych przy odwoływaniu się do swoich elementów. Oznacza to, że możesz odpowiednio indeksować zmienną `row` w przedstawionej powyżej pętli `for`, aby wybierać poszczególne elementy danych, które są Ci potrzebne. Zapis `row[2]` umożliwia na przykład wyłuskanie frazy, `row[3]` pozwala wydobyć litery, a `row[-1]` umożliwia wyciągnięcie samych wyników.

DB-API — krok 6. Zamknij swój kursor i połączenie

Po zapisaniu swoich danych w tabeli posprzątaj po sobie, zamykając kursor oraz połączenie.

```
>>> cursor.close()
True
>>> conn.close()
```

Sprzątanie po sobie jest zawsze dobrym pomysłem.

Zwróć uwagę, że powodzenie operacji zamknięcia kursora jest sygnalizowane zwróceniem wartości `True`, podczas gdy połączenie po prostu zostaje zamknięte. Zamknięcie kursora i połączenia, gdy nie są już potrzebne, jest zawsze dobrym pomysłem, ponieważ system bazodanowy dysponuje ograniczonym zbiorem zasobów. Programiści pracujący w laboratorium *Rusz głową!* lubią utrzymywać swoje kursory i połączenia bazodanowe otwarte tak długo, jak to potrzebne, ale ani chwili dłużej.

Zadanie nr 4: Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel

- ✔ Zainstaluj serwer bazy danych MySQL.
- ✔ Zainstaluj sterownik bazy danych MySQL dla języka Python.
- ✔ Utwórz bazę danych i tabele dla naszej aplikacji WWW.
- ✔ Utwórz kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Wykonaliśmy wszystkie zadania znajdujące się na naszej liście!

Po przejściu *sześciu kroków* przedstawionych w ramce *DB-API z bliska* dysponujesz kodem niezbędnym do komunikacji z tabelą `log`, co oznacza, że ukończyłeś zadanie nr 4 i utworzyłeś kod, za pomocą którego nasza aplikacja WWW będzie korzystała z bazy danych i tabel.

Przejrzyjmy kod, z którego możesz skorzystać (w całości). Został on zebrany poniżej.

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

import mysql.connector

conn = mysql.connector.connect(**dbconfig)

cursor = conn.cursor()

_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
          values
          (%s, %s, %s, %s, %s)"""

cursor.execute(_SQL, ('galaktyka', 'tym', '127.0.0.1', 'Opera', "{'t', 'y'}"))

conn.commit()

_SQL = """select * from log"""

cursor.execute(_SQL)

for row in cursor.fetchall():
    print(row)

cursor.close()

conn.close()
```

Zdefiniuj właściwości swojego połączenia.

Zaimportuj sterownik bazy danych.

Nawiąż połączenie i utwórz kursor.

Przypisz łańcuch znakowy zawierający zapytanie do zmiennej (zauważ obecność pięciu symboli zastępczych reprezentujących argumenty).

Wyślij zapytanie do serwera, pamiętając o podaniu wartości dla każdego wymaganego argumentu (w postaci krotki).

Wymuś na bazie zapis Twoich danych.

Pobierz z tabeli (zapisane przed chwilą) dane i wyświetl je na ekranie wiersz po wierszu.

Posprzątaj, gdy skończysz.

Skoro wykonałeś już wszystkie cztery zadania, jesteś gotów do modyfikacji kodu swojej aplikacji WWW w taki sposób, aby logowała dane żądań sieciowych w bazie MySQL zamiast w pliku tekstowym (jak dzieje się w tej chwili). Przystąpmy do tego teraz.

Magnesiki bazy danych

Przyjrzyj się raz jeszcze funkcji `log_request`, która została przedstawiona w poprzednim rozdziale.

Przypomnij sobie, że ta niewielka funkcja przyjmuje dwa argumenty: obiekt żądania sieciowego oraz wynik działania wyszukiwania.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```

Twoim zadaniem jest zastąpienie zestawu tej funkcji kodem, który będzie logował dane w Twojej bazie danych (zamiast w pliku tekstowym). Wiersz zawierający instrukcję `def` powinien pozostać bez zmian. Wybierz odpowiednie magnesiki spośród tych, które zostały rozrzucone w dolnej części tej strony, a następnie umieść je w odpowiednich miejscach, aby utworzyć działający kod funkcji.

```
def log_request(req: 'flask_request', res: str) -> None:
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................................
    ..............................
    ..............................
```

O rany! Co za bałagan! Czy pomożesz nam zaprowadzić tu porządek?

```
conn.close()
cursor = conn.cursor()
conn.commit()
_SQL = """select * from log"""
import mysql.connector
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }
_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
          values
          (%s, %s, %s, %s, %s)"""
for row in cursor.fetchall():
    print(row)
cursor.close()
cursor.execute(_SQL)
conn = mysql.connector.connect(**dbconfig)
```

Magnesiki bazy danych. Rozwiązanie

Miałeś raz jeszcze przyjrzeć się funkcji `log_request`, która została przedstawiona w poprzednim rozdziale.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```

Twoim zadaniem było zastąpienie zestawu tej funkcji kodem, który loguje dane w Twojej bazie danych (zamiast w pliku tekstowym). Wiersz zawierający instrukcję `def` miał pozostać bez zmian. Miałeś wybrać odpowiednie magnesiki spośród tych, które zostały rozrzucone w dolnej części strony.

```
def log_request(req: 'flask_request', res: str) -> None:
```

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }
```

Zdefiniuj właściwości połączenia.

```
import mysql.connector
```

```
conn = mysql.connector.connect(**dbconfig)
```

```
cursor = conn.cursor()
```

Zaimportuj sterownik, po czym nawiąż połączenie i utwórz kursor.

```
_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
          values
          (%s, %s, %s, %s, %s)"""
```

Utwórz łańcuch znakowy zawierający zapytanie, którego chcesz użyć.

```
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))
```

Wykonaj zapytanie.

To coś nowego: zamiast przechowywać cały łańcuch znakowy opisujący używaną przeglądarkę internetową (zapisany w req.user_agent), wydobywamy jedynie samą nazwę tej przeglądarki.

```
conn.commit()
```

```
cursor.close()
```

```
conn.close()
```

Po zapewnieniu, że dane zostały zapisane w bazie, sprzątamy po sobie, zamykając kursor i połączenie.

Te magnesiki nie były nam potrzebne.

```
cursor.execute(_SQL)
```

```
_SQL = """select * from log"""
```

```
for row in cursor.fetchall():
    print(row)
```

Jazda próbna

Zmień kod znajdujący się w Twoim pliku *vsearch4web.py*, zastępując oryginalny kod funkcji `log_request` tym, który został przedstawiony na poprzedniej stronie. Gdy już zapiszesz swój kod, uruchom najnowszą wersję swojej aplikacji WWW w wierszu poleceń. Pamiętaj, że w systemie *Windows* powinieneś użyć następującego polecenia:

```
C:\webapps> py -3 vsearch4web.py
```

Korzystając z systemu *Linux* lub *Mac OS X*, użyj z kolei polecenia:

```
$ python3 vsearch4web.py
```

Twoja aplikacja WWW powinna być dostępna pod następującym adresem:

http://127.0.0.1:5000/

Użyj przeglądarki internetowej do wykonania kilku wyszukiwań, aby potwierdzić, że Twoja aplikacja działa prawidłowo.

Chcielibyśmy zwrócić tu Twoją uwagę na dwie rzeczy:

- Twoja aplikacja WWW działa dokładnie tak samo, jak miało to miejsce wcześniej: każde wyszukiwanie powoduje wyświetlenie strony wyników.
- Użytkownicy Twojej aplikacji nie mają pojęcia, że dane wyszukiwania są teraz logowane w tabeli bazy danych zamiast w pliku tekstowym.

Niestety nie możemy teraz skorzystać z ciągu URL `/viewlog`, aby przejrzeć najnowsze wpisy logu, ponieważ funkcja związana z tym URL-em (`view_the_log`) współpracuje tylko z plikiem tekstowym *vsearch.log* (nie zaś z bazą danych). Wkrótce będziemy mieli nieco więcej do powiedzenia na temat możliwości poradzenia sobie z tym problemem.

Na tym etapie podsumujmy tę jazdę próbną, korzystając z konsoli MySQL w celu potwierdzenia, że nasza najnowsza wersja funkcji `log_request` loguje dane w tabeli `log`. Otwórz kolejne okno terminala i postępuj zgodnie z przedstawionymi poniżej wskazówkami (zwróć uwagę, że sformatowaliśmy i skróciliśmy nasze dane wyjściowe, aby mieściły się na stronie).

Zaloguj się do konsoli MySQL.

Za pomocą tego zapytania prosimy o wyświetlenie wszystkich danych znajdujących się w tabeli log (Twoje rzeczywiste dane prawdopodobnie będą się nieco różnić od naszych).

Administrator: Wiersz polecenia

```
C:\>mysql -u vsearch -p vsearchlogDB
Enter password: **************
Welcome to the MySQL monitor...

mysql> select * from log;
+----+---------------------+---------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase        | letters | ip        | browser_string | results              |
+----+---------------------+---------------+---------+-----------+----------------+----------------------+
|  1 | 2017-06-20 12:40:06 | autosto ... cz | aeiou   | 127.0.0.1 | firefox        | {'u', 'a', 'o', 'i'} |
|  2 | 2017-06-20 12:40:51 | mierni        | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2017-06-21 12:58:52 | galaktyka     | tym     | 127.0.0.1 | chrome         | {'y', 't'}           |
|  4 | 2017-06-21 13:02:14 | mierni        | vxy     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+---------------+---------+-----------+----------------+----------------------+
4 rows in set (0.00 sec)

mysql> quit
Bye
```

Nie zapomnij o opuszczeniu konsoli, gdy zakończysz pracę.

Pamiętaj, że tutaj przechowujemy już wyłącznie nazwę przeglądarki.

Przechowywanie danych to dopiero połowa sukcesu

Wykonawszy jazdę próbną z poprzedniej strony, masz już pewność, że Twój kod tworzący funkcję `log_request`, wykorzystujący DB-API Pythona, naprawdę przechowuje szczegóły każdego żądania sieciowego w tabeli `log`.

Raz jeszcze przyjrzyj się najnowszej wersji funkcji `log_request` (zawierającej tekst dokumentujący w pierwszym wierszu zestawu).

```
def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""
    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    import mysql.connector

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Doświadczeni programiści Pythona mogą wyrazić dezaprobatę po spojrzeniu na kod tej funkcji. Dowiesz się, dlaczego tak może być, z lektury jednej z kolejnych stron.

Ta nowa funkcja to wielka zmiana

Funkcja `log_request` zawiera znacznie więcej kodu niż wówczas, gdy korzystała z prostego pliku tekstowego, jednak ten dodatkowy kod jest potrzebny, aby komunikować się z bazą danych MySQL (z której będziesz korzystał na końcu tego rozdziału, aby odpowiedzieć na pytania dotyczące zalogowanych danych). Z tego powodu ten nowy, dłuższy i bardziej skomplikowany kod funkcji wydaje się tu usprawiedliwiony.

Pamiętaj też jednak, że Twoja aplikacja WWW ma jeszcze inną funkcję, o nazwie `view_the_log`, która pobiera dane z pliku logu *vsearch.log* i wyświetla je na ekranie w postaci ładnie sformatowanej strony WWW. Teraz powinieneś zaktualizować kod tej funkcji, aby radziła sobie z pobieraniem danych z należącej do bazy danych tabeli `log` zamiast z pliku tekstowego, jak miało to miejsce do tej pory.

Pytanie brzmi: jak najlepiej to zrobić?

Jak najlepiej ponownie użyć Twojego kodu obsługi bazy danych?

Masz już kod umożliwiający logowanie w bazie danych MySQL szczegółów każdego żądania obsługiwanego przez Twoją aplikację WWW. Napisanie czegoś podobnego w celu pobierania danych z tabeli `log` i wykorzystywania ich w funkcji `view_the_log` nie powinno wymagać zbyt wiele pracy. Powstaje jednak pytanie: jak zrobić to najlepiej? Zadaliśmy to pytanie trzem programistom... i otrzymaliśmy trzy zupełnie różne odpowiedzi.

Każda z powyższych sugestii jest na swój sposób słuszna, nawet jeśli niektóre są trochę podejrzane (zwłaszcza pierwsza z nich). Choć może się to okazać dla Ciebie pewną niespodzianką, prawdziwy programista Pythona raczej nie wybrałby żadnego z tych rozwiązań *z osobna*.

Zastanów się, co próbujesz ponownie wykorzystać

Spójrz raz jeszcze na nasz kod obsługi bazy danych należący do funkcji `log_request`.

Powinno być jasne, że niektóre fragmenty kodu tej funkcji możemy ponownie wykorzystać, pisząc dodatkowy kod odpowiedzialny za komunikację z bazą danych. Dlatego też postanowiliśmy skomentować kod funkcji, aby wskazać te części, które da się naszym zdaniem wykorzystać ponownie w odróżnieniu od tych, które są specyficzne dla głównego zadania wykonywanego przez funkcję `log_request`.

```
def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""
    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    import mysql.connector

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))

    conn.commit()
    cursor.close()
    conn.close()
```

Właściwości połączenia z bazą danych są bardzo specyficzne dla tego, co tu robimy, lecz prawdopodobnie będą potrzebne również w innych miejscach, więc powinno dać się ich ponownie użyć.

Te dwie instrukcje zawsze będą takie same, dlatego mogą być wykorzystane ponownie.

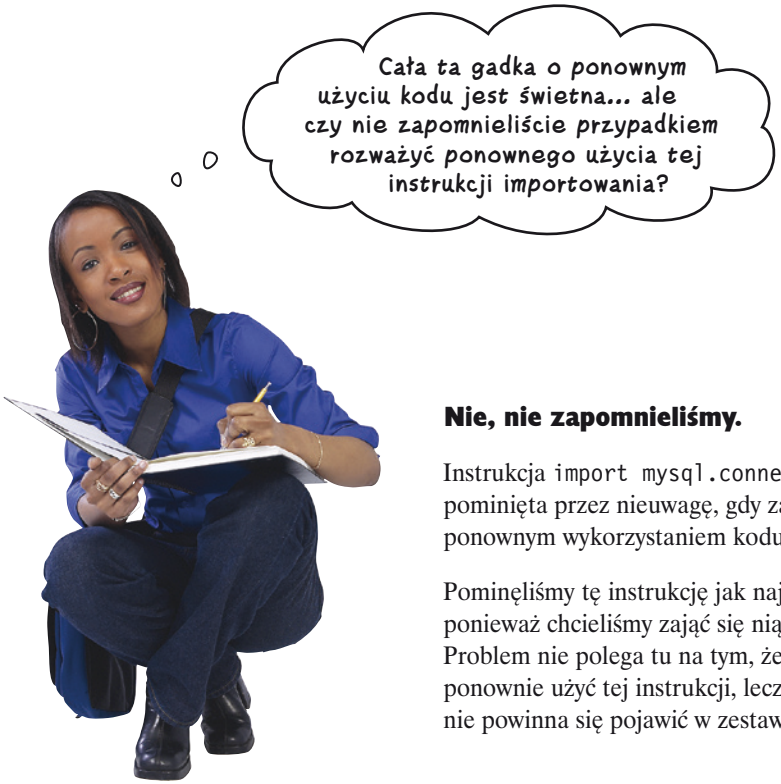
Te trzy instrukcje również będą takie same, dlatego też mogą być wykorzystane ponownie.

Ten kod stanowi prawdziwe „bebechy" tego, co się dzieje wewnątrz naszej funkcji, i nie da się go ponownie użyć w żaden sensowny sposób (jako że jest zdecydowanie zbyt specyficzny dla przeprowadzanej operacji).

Na podstawie tej prostej analizy można wyróżnić w funkcji `log_request` trzy grupy instrukcji kodu:

- instrukcje, które da się łatwo ponownie wykorzystać (takie jak te, które są odpowiedzialne za tworzenie zmiennych `conn` i `cursor` oraz wywołania metod `commit` i `close`);
- instrukcje, które są specyficzne dla problemu, lecz pomimo to powinno je się dać ponownie wykorzystać;
- instrukcje, których nie da się ponownie wykorzystać (takie jak przypisanie do zmiennej `_SQL` czy wywołanie metody `cursor.execute`). Wszystkie kolejne interakcje z bazą danych MySQL będą niemal na pewno wymagać zupełnie innych zapytań SQL, jak również zupełnie innych argumentów (jeśli w ogóle będą jakieś potrzebne).

A co z tą instrukcją importowania?



Nie, nie zapomnieliśmy.

Instrukcja `import mysql.connector` nie została pominięta przez nieuwagę, gdy zastanawialiśmy się nad ponownym wykorzystaniem kodu funkcji `log_request`.

Pominęliśmy tę instrukcję jak najbardziej celowo, ponieważ chcieliśmy zająć się nią w specjalny sposób. Problem nie polega tu na tym, że nie zamierzamy ponownie użyć tej instrukcji, lecz na tym, że w ogóle nie powinna się pojawić w zestawie funkcji!

Zachowaj ostrożność, umieszczając swoje instrukcje importowania

Kilka stron wcześniej wspomnieliśmy, że doświadczeni programiści Pythona mogą wyrazić dezaprobatę, spojrzawszy na kod funkcji `log_request`. Powodem jest umieszczenie w jej zestawie wiersza `import mysql.connector`. Dezaprobata ta będzie uzasadniona pomimo tego, że nasza ostatnia jazda próbna jasno wykazała, że kod ten działa prawidłowo. W czym więc problem?

Problem ma związek z tym, co się dzieje, gdy interpreter napotyka w Twoim kodzie instrukcję `import`: zaimportowany moduł jest odczytywany w całości, po czym wykonywany przez interpreter. Działanie to nie sprawia nam kłopotu, gdy odbywa się *poza funkcją*, ponieważ importowany moduł jest (przeważnie) odczytywany tylko *jeden raz*, a następnie tylko *jeden raz* wykonywany.

Jednak jeśli instrukcja `import` występuje *w obrębie* funkcji, jest odczytywana *oraz* wykonywana **za każdym razem, gdy funkcja zostaje wywołana**. Uważa się to za ekstremalnie marnotrawną praktykę (mimo że, jak miałeś okazję się przekonać, interpreter nie powstrzyma Cię przed umieszczeniem instrukcji importowania wewnątrz funkcji). Mamy dla Ciebie bardzo prostą radę: dobrze się zastanów nad tym, gdzie umieszczasz swoje instrukcje `import`, i nie wstawiaj żadnej z nich w kodzie funkcji.

Zastanów się, co próbujesz zrobić

Poza przeprowadzeniem analizy kodu funkcji `log_request` pod względem możliwości jego ponownego wykorzystania można też spojrzeć na ten kod z perspektywy tego, *kiedy* jest wykonywany.

„Bebechy" tej funkcji stanowi przypisanie zapytania do zmiennej `_SQL` oraz wywołanie metody `cursor.execute`. Te dwie instrukcje najbardziej ewidentnie reprezentują *to*, co funkcja powinna **wykonywać** i co jest tu — szczerze mówiąc — rzeczą najważniejszą. Za pomocą pierwszych instrukcji funkcji definiujemy właściwości połączenia (zebrane w zmiennej `dbconfig`), a następnie tworzymy połączenie i kursor. Ten kod **konfiguracyjny** zawsze musi zostać wykonany *przed* zasadniczą częścią funkcji. Ostatnie trzy instrukcje funkcji (jedno wywołanie metody `commit` i dwa wywołania metody `close`) są wykonywane *po* tym, jak zostanie wykonana zasadnicza część funkcji. Jest to kod **demontażowy**, odpowiadający za wszelkie czynności związane z niezbędnym sprzątaniem.

Biorąc pod uwagę ten wzorzec *konfiguracji*, *wykonania* i *demontażu*, przyjrzyjmy się naszej funkcji raz jeszcze. Zauważ, że przenieśliśmy instrukcję `import` w miejsce, w którym będzie wykonywana poza zestawem funkcji `log_request` (aby uniknąć wszelkich dalszych jęków dezaprobaty).

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""

    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))

    conn.commit()
    cursor.close()
    conn.close()
```

To zdecydowanie lepsze miejsce dla wszelkich instrukcji importowania (powinny się one znajdować poza zestawem funkcji).

To jest kod konfiguracyjny, który jest wykonywany przed tym, zanim funkcja wykona swoje zasadnicze zadanie.

Ten kod odpowiada za to, co funkcja FAKTYCZNIE robi, czyli za logowanie żądań sieciowych w bazie danych.

To kod demontażowy, który wykonuje się już po tym, jak funkcja spełniła swoje zasadnicze zadanie.

Czy nie byłoby dobrze, gdyby istniał jakiś sposób, który pozwalałby ponownie wykorzystać ten wzorzec konfiguracji, wykonania i demontażu?

Miałeś już wcześniej do czynienia z tym wzorcem

Przeanalizuj ten wzorzec, który przed chwilą wskazaliśmy: przygotowanie kodu do pracy, po którym następuje kod wykonujący niezbędne zadanie, a po nim kod odpowiedzialny za posprzątanie po wszystkim. Może nie być to oczywiste od razu, ale w poprzednim rozdziale robiliśmy już coś podobnego, korzystając z kodu, który pasuje do tego wzorca. Poniżej został on zaprezentowany raz jeszcze.

Otwórz plik.

Przypisz strumień pliku do zmiennej.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

Wykonaj jakieś operacje (ten fragment kodu jest taki sam jak powyżej).

Przypomnij sobie, jak instrukcja `with` *zarządza kontekstem*, w którym jest wykonywany kod należący do jej zestawu. Gdy korzystasz z plików (jak miało to miejsce w przypadku kodu przedstawionego powyżej), instrukcja ta odpowiada za otwarcie pliku o podanej nazwie i zwrócenie zmiennej reprezentującej odpowiedni strumień pliku. W tym przykładzie jest to zmienna `tasks`, a ten fragment stanowi kod **konfiguracyjny**. Zestaw związany z instrukcją `with` to kod odpowiedzialny za **wykonanie** zadania; w naszym przypadku jest to pętla `for`, która załatwia rzeczywistą pracę (czyli „ważną część"). Wreszcie, gdy korzystasz z instrukcji `with` do otwarcia pliku, gwarantuje Ci ona, że plik ten zostanie zamknięty, gdy zestaw tej instrukcji się zakończy. To jest kod **demontujący**.

Byłoby cudownie, gdyby dało się zintegrować nasz kod obsługi bazy danych z instrukcją `with`. Byłoby wręcz idealnie, gdybyśmy mogli napisać kod przedstawiony poniżej i pozostawić instrukcji `with` wszelkie troski związane z konfiguracją bazy danych oraz szczegółami demontażu.

Nadal musimy zdefiniować właściwości połączenia.

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

with UseDatabase(dbconfig) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Ta instrukcja with operuje na bazach danych zamiast na plikach dyskowych i zwraca kursor, z którego możemy korzystać.

Przedstawiony na poprzedniej stronie kod odpowiedzialny za wykonanie zadania pozostaje bez zmian.

Nie próbuj jeszcze uruchamiać tego kodu, ponieważ musisz przedtem napisać kod menedżera kontekstu UseDatabase.

Dobra wiadomość jest taka, że Python zapewnia **protokół zarządzania kontekstem**, który umożliwia programistom korzystanie z instrukcji `with` zgodnie z ich potrzebami. To z kolei prowadzi do *złej wiadomości*...

Zła wiadomość nie jest wcale aż taka zła

Na końcu ostatniej strony stwierdziliśmy, że *dobrą wiadomością* jest to, iż Python zapewnia protokół zarządzania kontekstem, który umożliwia programistom korzystanie z instrukcji `with` wtedy, gdy jest to konieczne. Gdy dowiesz się, jak to robić, będziesz w stanie utworzyć menedżer kontekstu o nazwie `UseDatabase`, z którego następnie będziesz mógł korzystać w instrukcji `with` w celu komunikowania się z Twoją bazą danych.

Pomysł polega tu na tym, aby cały ten nudny i powtarzalny, związany z konfiguracją i demontażem kod, który właśnie napisałeś, aby zapisać logowane dane w bazie, zastąpić pojedynczą instrukcją `with`, która wygląda tak, jak przedstawiona poniżej.

```
    ...
with UseDatabase(dbconfig) as cursor:
    ...
```

Ta instrukcja with przypomina tę, której używaliśmy w przypadku plików i wbudowanej funkcji open. Różnica polega na tym, że instrukcja używana tutaj operuje na bazie danych zamiast na pliku.

Złą wiadomością jest to, że zadanie polegające na utworzeniu menedżera kontekstu komplikuje fakt, iż aby skutecznie podłączyć się do protokołu, musisz wiedzieć, jak tworzy się klasę Pythona.

Zauważ, że aż do tej chwili w książce tej udało nam się napisać mnóstwo użytecznego kodu bez konieczności tworzenia choćby jednej klasy, co jest całkiem niezłym wynikiem, szczególnie gdy weźmie się pod uwagę to, że niektóre języki programowania nie pozwalają zrobić *czegokolwiek* bez wcześniejszego utworzenia klasy (mamy na myśli *Ciebie*, Javo).

Teraz jednak nadszedł czas, aby wziąć byka za rogi (chociaż, jeśli mamy być szczerzy, tworzenie klasy w Pythonie nie jest czymś, czego należy się obawiać).

Ponieważ umiejętność tworzenia klas jest czymś ogólnie przydatnym, odejdźmy na razie od naszego bieżącego tematu dotyczącego dodawania kodu obsługi bazy danych do naszej aplikacji WWW i cały następny (krótki) rozdział poświęćmy klasom. Przedstawimy w nim tylko to, co jest niezbędne, aby umożliwić Ci utworzenie menedżera kontekstu `UseDatabase`. Gdy uda nam się to zrobić, w kolejnym rozdziale wrócimy do naszego kodu obsługi bazy danych (i aplikacji WWW) oraz wykorzystamy nowo nabyte umiejętności pisania klas do opracowania kodu menedżera kontekstu `UseDatabase`.

Kod przykładów z rozdziału 7.

To jest kod obsługi bazy danych, z którego w tej chwili korzystamy w naszej aplikacji WWW (tzn. funkcja log_request).

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""

    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

with UseDatabase(dbconfig) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

To jest kod, z którego chcielibyśmy skorzystać, aby zapewnić wykonanie tej samej operacji, jaką wykonuje bieżący kod (zastępując w ten sposób obecny zestaw funkcji log_request). Nie powinieneś jednak jeszcze próbować go uruchamiać, ponieważ nie będzie on działał bez menedżera kontekstu UseDatabase.

Łączenie zachowania ze stanem

Klasy umożliwiają łączenie kodu odpowiedzialnego za stan i działanie.

W tym rozdziale na jakiś czas zostawisz swoją aplikację WWW w spokoju i zajmiesz się nauką tworzenia **klas** w języku Python. Zrobisz to, aby uzyskać możliwość utworzenia menedżera kontekstu, w czym pomoże Ci właśnie klasa Pythona. Rozdział ten poświęcamy w całości kwestiom związanym z tworzeniem i używaniem klas, ponieważ są to rzeczy, które i tak zdecydowanie warto znać. Nie omówimy tu wszystkich tematów dotyczących klas, ale poruszymy wszystkie zagadnienia, jakie musisz zrozumieć, aby bez kłopotu utworzyć menedżer kontekstu, którego oczekuje Twoja aplikacja. Zacznijmy więc i zobaczmy, z czym to się wiąże.

Zastosowanie instrukcji with

Jak stwierdziliśmy już pod koniec poprzedniego rozdziału, sposób włączenia kodu konfiguracyjnego i demontującego do instrukcji `with` języka Python jest całkiem prosty... jeśli tylko się wie, jak utworzyć **klasę** Pythona.

Mimo że przeczytałeś już ponad połowę tej książki, jak dotąd radziłeś sobie doskonale bez konieczności zdefiniowania choćby jednej klasy. Napisałeś w międzyczasie całkiem przydatny kod — który może być ponownie użyty — korzystając przy tym wyłącznie z możliwości funkcji zapewnianych przez język Python. Istnieją jednak również inne sposoby pisania i porządkowania kodu, a orientacja obiektowa jest jedną z najpopularniejszych.

Korzystając z Pythona, nigdy nie jesteś zmuszony programować wyłącznie zgodnie z paradygmatem zorientowanym obiektowo, a język ten oferuje Ci dużą elastyczność w kwestii sposobu pisania kodu. Jeśli jednak chodzi o korzystanie z instrukcji `with`, zastosowanie klasy jest **zalecanym podejściem**, mimo że biblioteka standardowa zapewnia obsługę podobnych możliwości *bez* konieczności jej używania (jednak sposób ten ma o wiele węższe zastosowanie, dlatego nie zamierzamy tu z niego korzystać).

Aby zatem użyć instrukcji `with`, będziesz musiał utworzyć klasę. Gdy będziesz już wiedział, jak pisać klasy, będziesz mógł utworzyć tę, która implementuje **protokół zarządzania kontekstem** i spełnia jego wymogi. Protokół ten stanowi (wbudowany w Pythona) mechanizm umożliwiający korzystanie z instrukcji `with`.

Nauczmy się tworzyć klasy i używać ich w języku Python, zanim w następnym rozdziale powrócimy do omawiania protokołu zarządzania kontekstem.

Protokół zarządzania kontekstem umożliwia pisanie klas, które łączą się z instrukcją with.

Nie istnieją głupie pytania

P: Jakim dokładnie językiem programowania jest Python: zorientowanym obiektowo, funkcyjnym czy proceduralnym?

O: To doskonałe pytanie, które prędzej czy później zadaje wielu programistów przesiadających się na język Python. Odpowiedź jest taka, że Python wspiera zastosowanie paradygmatów programowania zapożyczonych ze wszystkich tych trzech popularnych podejść i zachęca programistów do ich mieszania zgodnie z bieżącymi potrzebami. Koncepcję tę może być Ci dość trudno przyswoić, zwłaszcza jeśli do tej pory korzystałeś z takich rozwiązań, że cały pisany przez Ciebie kod musiał znajdować się w klasie, której obiekty tworzyłeś (jak ma to miejsce w innych językach programowania, takich jak Java).

Radzimy Ci się tym szczególnie nie przejmować i tworzyć kod zgodnie z dowolnym paradygmatem, który Ci najbardziej odpowiada, i nie rezygnować jednak z zastosowania innych paradygmatów tylko dlatego, że — jako alternatywne podejścia — wydają Ci się obce.

P: A zatem... czy niedobrze jest zaczynać od utworzenia klasy?

O: Nie, nie jest, jeśli tylko tego wymaga Twoja aplikacja. Nie musisz umieszczać całego swojego kodu w klasach, jeśli jednak postanowisz to zrobić, Python nie będzie Ci w tym przeszkadzał.

Jak do tej pory w książce tej udało nam się uniknąć konieczności tworzenia klas, ale dotarliśmy właśnie do punktu, w którym sensowne jest zastosowanie klasy w celu rozwiązania występującego w naszej aplikacji WWW specyficznego problemu, z jakim się borykamy. To całkowicie w porządku, że swobodnie mieszamy różne paradygmaty programowania, aby znaleźć rozwiązanie naszego obecnego problemu.

Elementarz programowania zorientowanego obiektowo

Zanim zaczniemy korzystać z klas, chcemy wyraźnie zaznaczyć, że nie zamierzamy w tym rozdziale wyczerpać wszystkich tematów związanych z klasami w języku Python. Pragniemy jedynie pokazać Ci tyle, abyś mógł swobodnie utworzyć klasę implementującą protokół zarządzania kontekstem.

Z tego też powodu nie będziemy tu omawiać pewnych zagadnień, które doświadczeni praktycy programowania zorientowanego obiektowo (ang. *object-oriented programming*, *OOP*) prawdopodobnie spodziewaliby się w tym miejscu znaleźć. Są wśród nich takie tematy jak *dziedziczenie* (ang. *inheritance*) i *polimorfizm* (chociaż Python radzi sobie z obsługą obydwu tych rzeczy). Jest tak, ponieważ tworząc menedżer kontekstu, będziemy zainteresowani przede wszystkim **hermetyzacją** (ang. *encapsulation*).

Jeśli słownictwo użyte w poprzednim akapicie wpędziło Cię w popłoch, nie martw się, ponieważ możesz spokojnie kontynuować lekturę bez znajomości znaczenia któregokolwiek z tych pojęć.

Na podstawie treści ostatniej strony wiesz, że musisz utworzyć klasę, aby skorzystać z instrukcji `with`. Zanim przejdziemy do szczegółów dotyczących sposobu, w jaki się to robi, przyjrzymy się — pisząc kod przykładowej klasy w trakcie omawiania tematu — temu, co tworzy klasę w Pythonie. Gdy zrozumiesz już, jak napisać klasę, powrócimy do zastosowania instrukcji `with` (w kolejnym rozdziale).

Nie daj się przestraszyć tym wszystkim widocznym obok dziwacznym słowom!

Gdybyśmy mieli wyłonić zwycięzcę w konkursie na należącą do tej książki stronę, która zawiera najwięcej magicznych słów, ta wygrałaby w cuglach. Nie zniechęcaj się jednak całym tym żargonem, który się tu pojawił. Jeśli znasz już paradygmat OOP, wszystkie te słowa powinny mieć dla Ciebie sens. **W przeciwnym razie — naprawdę ważne kwestie przedstawione są poniżej.** Nie martw się, ponieważ wszystko będzie się wyjaśniać wraz z tym, jak będziesz się zapoznawał z przykładami zaprezentowanymi na kilku następnych stronach.

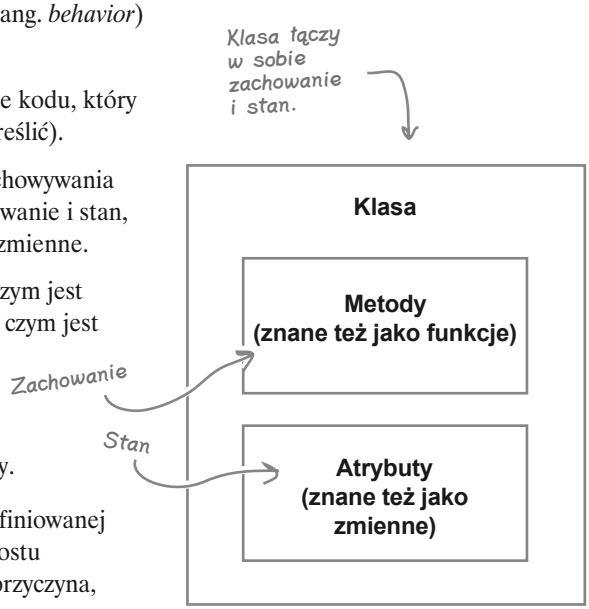
Klasa łączy w sobie zachowanie i stan

Zastosowanie klasy umożliwia Ci połączenie ze sobą **zachowania** (ang. *behavior*) i **stanu** (ang. *state*) w jednym obiekcie.

Gdy napotkasz słowo *zachowanie*, pomyśl o *funkcji*, czyli fragmencie kodu, który coś robi (lub *implementuje pewne zachowanie*, jeśli tak wolisz to określić).

Gdy usłyszysz słowo *stan*, pomyśl o *zmiennej*, czyli miejscu do przechowywania wartości w ramach klasy. Gdy piszemy, że klasa łączy *ze sobą* zachowanie i stan, mamy na myśli po prostu to, że klasa upakowuje w sobie funkcje i zmienne.

Z powyższych stwierdzeń płynie jeden prosty wniosek: jeśli wiesz, czym jest funkcja i czym są zmienne, jesteś na dobrej drodze, aby zrozumieć, czym jest klasa (a także dowiedzieć się, jak ją utworzyć).



Klasy mają metody i atrybuty

W języku Python zachowanie klasy definiuje się, tworząc jej metody.

Słowo *metoda* to używana w paradygmacie OOP nazwa funkcji zdefiniowanej wewnątrz klasy. Powód, dla którego metody nie są nazywane po prostu *funkcjami klasy*, zaginął w pomroce dziejów, podobnie zresztą jak przyczyna, dla której nie używa się określenia *zmienne klasy*, lecz nazwy *atrybuty*.

Tworzenie obiektów klas

Aby skorzystać z klasy, tworzy się jej obiekt (przykład tej operacji zobaczysz poniżej). Działanie to określa się mianem **tworzenia instancji obiektu** (ang. *object instantiation*). Gdy usłyszysz o *tworzeniu instancji*, pomyśl o *wywoływaniu*; chodzi o to, że wywołujesz klasę, aby utworzyć obiekt.

Może Ci się to wydawać dość zaskakujące, ale możesz utworzyć klasę, która nie ma stanu ani zachowania, nadal stanowi jednak pełnoprawną klasę w języku Python. Klasa taka jest *pusta*. Zacznijmy nasze eksperymenty od pustej klasy, którą będziemy rozbudowywać na kolejnych stronach tego rozdziału. Będziemy tu korzystać z wiersza poleceń interpretera i zachęcamy Cię, abyś robił to samo.

Rozpoczniemy od utworzenia pustej klasy o nazwie `CountFromBy`. Najpierw umieścimy przed tą nazwą słowo kluczowe `class`, a następnie podamy zestaw kodu, który implementuje klasę (po obowiązkowym znaku dwukropka).

Definicje klas rozpoczynają się słowem kluczowym class.

Nie zapomnij o dwukropku.

```
>>> class CountFromBy:
        pass
```

Tutaj znajduje się zestaw klasy.

To jest nazwa klasy.

Instrukcja pass jest prawidłowa (tzn. jest syntaktycznie poprawna), ale nie wykonuje żadnej operacji. Traktuj ją jako pustą instrukcję.

Zwróć uwagę na fakt, że przedstawiony powyżej zestaw klasy zawiera jedynie słowo kluczowe `pass` języka Python, stanowiące pustą instrukcję (w tym sensie, że nie wykonuje ona żadnego działania). Słowa tego możesz używać wszędzie tam, gdzie interpreter spodziewa się zastać faktyczny kod. W tym przypadku nie jesteśmy jeszcze gotowi podać szczegółów klasy `CountFromBy`, dlatego skorzystaliśmy ze słowa kluczowego `pass`, aby uniknąć wystąpienia błędów składni, które normalnie zostałyby zgłoszone, gdybyśmy spróbowali utworzyć klasę z zestawem całkowicie pozbawionym kodu.

Teraz, gdy nasza klasa już istnieje, utwórzmy jej dwa obiekty, z których jeden będzie się nazywał `a`, natomiast drugi — `b`. Zauważ, że tworzenie obiektów klasy bardzo przypomina operację wywołania funkcji, co pokazano poniżej.

```
>>> a = CountFromBy()
>>> b = CountFromBy()
```

Te instrukcje bardzo przypominają wywołania funkcji, nieprawdaż?

Utwórz obiekt: dodaj do nazwy klasy nawias okrągły, po czym całość przypisz do zmiennej.

Nie istnieją głupie pytania

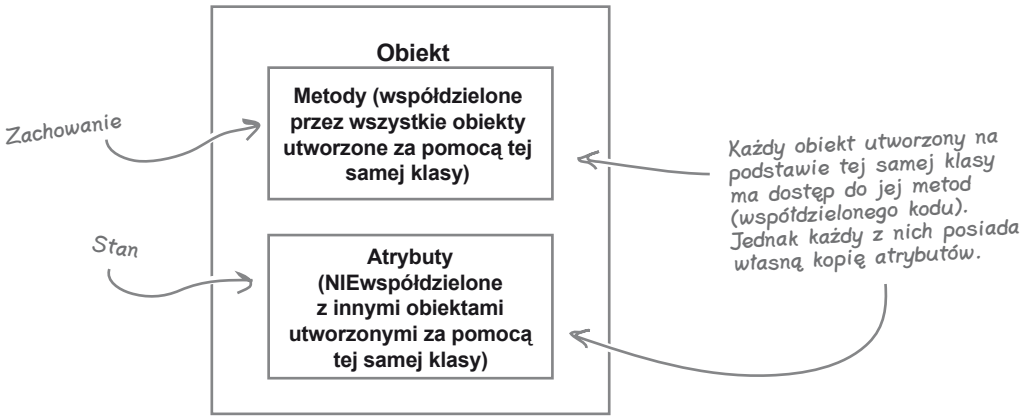
P: Gdy patrzę w cudzy kod, skąd mam wiedzieć, czy coś takiego jak `CountFromBy()` stanowi instrukcję tworzenia obiektu czy też wywołanie funkcji? Wygląda mi to raczej na wywołanie funkcji...

O: To świetne pytanie. Na pierwszy rzut oka rzeczywiście możesz nie wiedzieć. Wśród programistów Pythona funkcjonuje jednak ogólnie przyjęta konwencja, zgodnie z którą w nazwach funkcji używa się małych liter (oraz znaków podkreślenia do rozdzielania poszczególnych słów), podczas gdy notacja *CamelCase*[1] (połączone ze sobą słowa, których pierwsze litery pisane są jako wielkie) wykorzystywana jest do nazywania klas. Zgodnie z tą konwencją powinno być jasne, że `count_from_by()` to wywołanie funkcji, a `CountFromBy()` stanowi instrukcję tworzenia obiektu. Wszystko jest w porządku, dopóki wszyscy trzymają się tej konwencji, co **gorąco polecamy** robić też Tobie. Jeśli jednak zignorujesz tę sugestię, wszystko jest możliwe, a większość programistów Pythona będzie się z pewnością starała unikać zarówno Ciebie, jak i Twojego kodu.

[1] Ściślej: jest to notacja *UpperCamelCase* (określana też jako *PascalCase*), ponieważ również pierwsze słowo nazwy pisane jest tu wielką literą (nazwa w notacji *camelCase* zaczyna się małą literą) — *przyp. tłum.*

Obiekty współdzielą zachowanie, ale nie stan

Gdy tworzysz obiekty klasy, każdy z nich współdzieli zakodowane zachowania klasy (czyli metody, które zostały w niej zdefiniowane), ale posiada własną kopię wszystkich stanów (czyli atrybutów), tak jak zostało to pokazane poniżej.



Ta różnica nabierze sensu, gdy rozwiniemy nieco nasz przykład z klasą `CountFromBy`.

Definiowanie tego, czego oczekujemy od klasy CountFromBy

Zdefiniujmy teraz, co klasa `CountFromBy` ma tak naprawdę według nas robić (ponieważ pusta klasa rzadko się do czegoś przydaje).

Zwróć uwagę, że ta nowa klasa CountFromBy jeszcze nie istnieje. Utworzysz ją jednak już wkrótce.

Sprawmy, aby klasa `CountFromBy` reprezentowała licznik, który można inkrementować. Standardowo licznik ten będzie zaczynał zliczanie od wartości 0 i będzie ją zwiększał o 1. Umożliwimy również określenie innej wartości początkowej lub wartości, o którą ma się zwiększać licznik. Oznacza to, że będziesz na przykład mógł utworzyć obiekt klasy `CountFromBy`, który zaczyna zliczanie od wartości 100 i zwiększa swoją wartość o 10.

Sprawdźmy, co nasza klasa `CountFromBy` będzie w stanie nam zapewnić (gdy opracujemy już jej kod). Dzięki zrozumieniu sposobu, w jaki klasa ta będzie używana, łatwiej będzie Ci zrozumieć jej kod, gdy będziemy go pisać. W naszym pierwszym przykładzie skorzystamy ze standardowych wartości używanych przez klasę: licznik zacznie pracę od wartości 0 i będzie zwiększał swoją wartość o 1, gdy zostanie o to poproszony za pomocą wywołania metody `increase`. Nowo utworzony obiekt zostaje tu przypisany do nowej zmiennej, której nadaliśmy nazwę `c`.

```
>>> c = CountFromBy()
>>> c
0
>>> c.increase()
>>> c.increase()
>>> c.increase()
>>> c
3
```

Utwórz kolejny nowy obiekt i przypisz go do zmiennej o nazwie „c".

Wartość początkowa wynosi 0.

Wywołaj metodę increase, aby zwiększyć wartość licznika za każdym razem o 1.

Po trzech wywołaniach metody increase wartość obiektu wynosi 3.

Więcej działań z klasą CountFromBy

Przykładowe użycie klasy `CountFromBy` z końca poprzedniej strony pokazuje jej standardowe zachowanie polegające na tym, że gdy wartości nie są jawnie określone, licznik reprezentowany przez klasę zaczyna swoje działanie od wartości 0 i zwiększa ją zawsze o 1. Możliwe jest także podanie alternatywnej wartości początkowej, tak jak zostało to pokazane w następnym przykładzie zamieszczonym poniżej, w którym licznik zaczyna pracę od wartości 100.

```
>>> d = CountFromBy(100)
>>> d
100
>>> d.increase()
>>> d.increase()
>>> d.increase()
>>> d
103
```

Tworząc kolejny nowy obiekt, określ wartość początkową.

Wartość początkowa wynosi 100.

Wywołaj metodę increase, aby zwiększyć wartość licznika za każdym razem o 1.

Po trzech wywołaniach metody increase wartość przechowywana przez obiekt d wynosi 103.

Oprócz wartości początkowej można też określić wartość, o którą zwiększany będzie licznik, tak jak zostało to przedstawione poniżej, gdzie zaczęliśmy zliczanie od 100 i zwiększamy wartość o 10.

```
>>> e = CountFromBy(100, 10)
>>> e
100
>>> for i in range(3):
        e.increase()
>>> e
130
```

Tu określamy zarówno wartość początkową, jak i liczbę, o którą należy zwiększyć naszą wartość.

Wartość początkowa obiektu przechowywanego w zmiennej e wynosi 100, a końcowa to 130.

Wywołaj metodę increase trzykrotnie w pętli for, za każdym razem zwiększając wartość licznika o 10.

W ostatnim z przedstawionych tu przykładów licznik zaczyna swoje działanie od wartości 0 (będącej wartością domyślną), ale jest zwiększany o 15. Zamiast podawać wartości `(0, 15)` jako argumenty klasy, skorzystaliśmy w tym miejscu ze słowa kluczowego argumentu, które umożliwia nam określenie wartości, o jaką należy zwiększyć licznik, pozostawiając jednak standardową wartość początkową (czyli 0).

```
>>> f = CountFromBy(increment=15)
>>> f
0
>>> for j in range(3):
        f.increase()
>>> f
45
```

Tu określamy liczbę, o którą należy zwiększyć naszą wartość.

Wartość początkowa obiektu przechowywanego w zmiennej f wynosi 0, a końcowa to 45.

Tak jak wcześniej, wywołujemy funkcję increase trzykrotnie.

Warto się powtarzać: obiekty współdzielą zachowanie, ale nie stan

W poprzednich przykładach utworzyliśmy cztery nowe obiekty klasy `CountFromBy`: `c`, `d`, `e` oraz `f`, z których każdy ma dostęp do metody `increase` definiującej zachowanie współdzielone przez wszystkie obiekty utworzone na podstawie tej klasy. Zawsze będzie istniała tylko jedna kopia kodu tej metody, z której korzystać będą wszystkie te obiekty. Każdy z nich posiada jednak własne wartości atrybutów. W naszym przykładzie jest to wartość licznika, która jest inna dla każdego z obiektów.

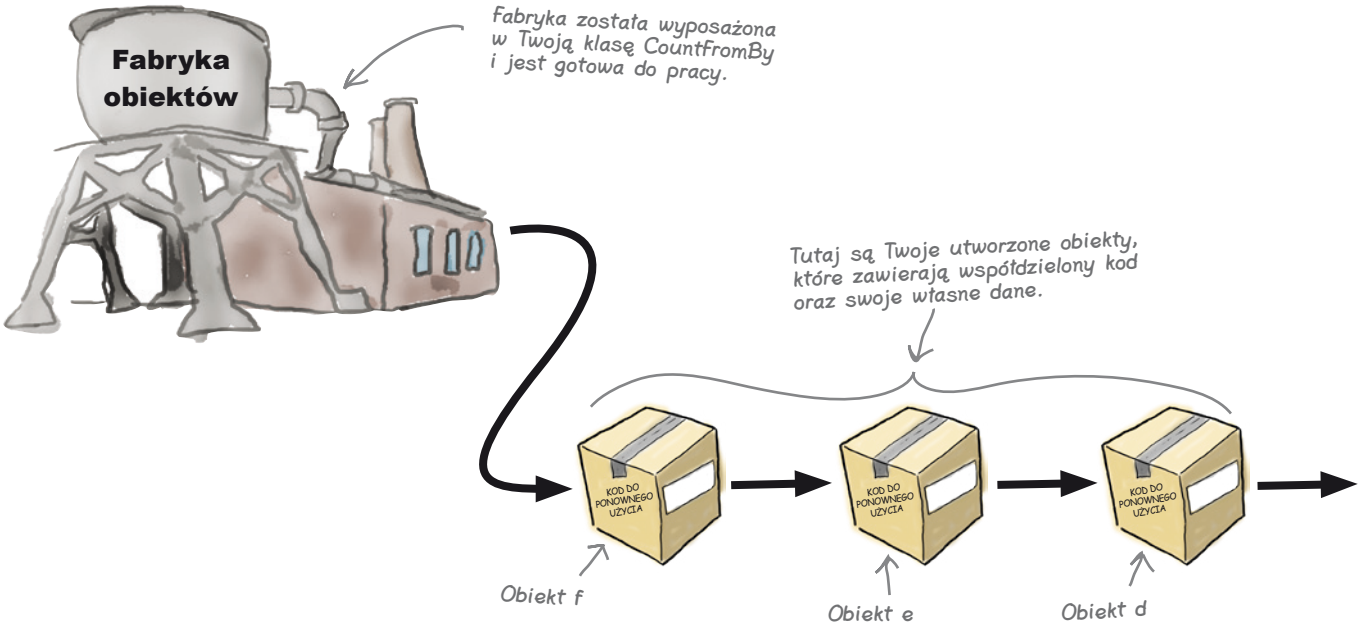
```
>>> c
3
>>> d
103
>>> e
130
>>> f
45
```

Te cztery obiekty klasy CountFromBy mają własne wartości atrybutów.

Zachowanie klasy jest współdzielone przez każdy jej obiekt, podczas gdy stan nie jest współdzielony. Każdy obiekt posiada własny stan.

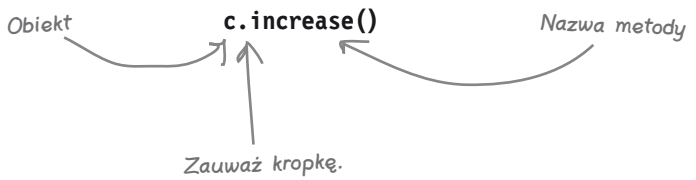
Raz jeszcze powtórzymy tu tę kluczową kwestię: kod metody jest współdzielony, ale dane atrybutów nie są.

O klasach można myśleć jako o „szablonach do wycinania ciastek”, które są używane w fabryce do masowej produkcji obiektów zachowujących się w ten sam sposób, ale przechowujących własne dane.

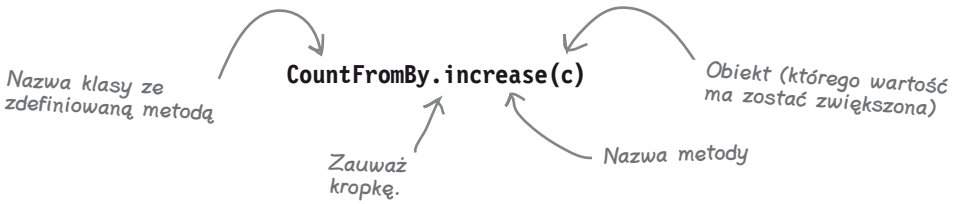


Wywołanie metody: zrozumienie szczegółów

Stwierdziliśmy wcześniej, że metoda to *funkcja zdefiniowana wewnątrz klasy*. Przedstawiliśmy również przykłady wywołania metody należącej do klasy `CountFromBy`. Metodę `increase` wywołuje się, używając dobrze znanej Ci notacji kropkowej.



Pouczające jest rozważenie tego, jaki kod jest *naprawdę* wykonywany (w tle) przez interpreter, gdy napotka on zaprezentowany powyżej wiersz kodu. Oto wywołanie, w które *zawsze* zmienia on powyższy wiersz. Zwróć uwagę na to, co się dzieje ze zmienną c.



Tak, dokładnie. Ale nikt nigdy tego nie robi.

I nie powinieneś być tu wyjątkiem, ponieważ interpreter Pythona i tak wykonuje to za Ciebie, zatem... po co pisać więcej kodu, aby zrobić coś, co dałoby się osiągnąć za pomocą krótszego zapisu?

Powód, dla którego interpreter to robi, stanie się jasny, gdy dowiesz się więcej o tym, jak działają metody.

Wywołanie metody: co się faktycznie dzieje

Na pierwszy rzut oka to, że interpreter przekształca konstrukcję `c.increase()` w `CountFromBy.increase(c)`, może wyglądać dość dziwnie, ale przyjęcie do wiadomości tego, że się to dzieje, pomaga zrozumieć, dlaczego każda metoda, którą piszesz, przyjmuje *co najmniej* jeden argument.

Oczywiście nic nie stoi na przeszkodzie, aby metody przyjmowały więcej argumentów, ale pierwszy z nich *zawsze* musi istnieć, aby przekazywać obiekt, na rzecz którego należy wykonać działanie (tym obiektem w przykładzie z poprzedniej strony jest `c`). W gruncie rzeczy szeroko rozpowszechnioną praktyką w społeczności programistów Pythona jest nadawanie pierwszemu argumentowi każdej metody specjalnej nazwy: `self`.

Gdy metoda `increase()` jest wywoływana jako `c.increase()`, możesz sobie wyobrażać, że jej wiersz `def` powinien wyglądać tak jak poniżej.

```
def increase():
```

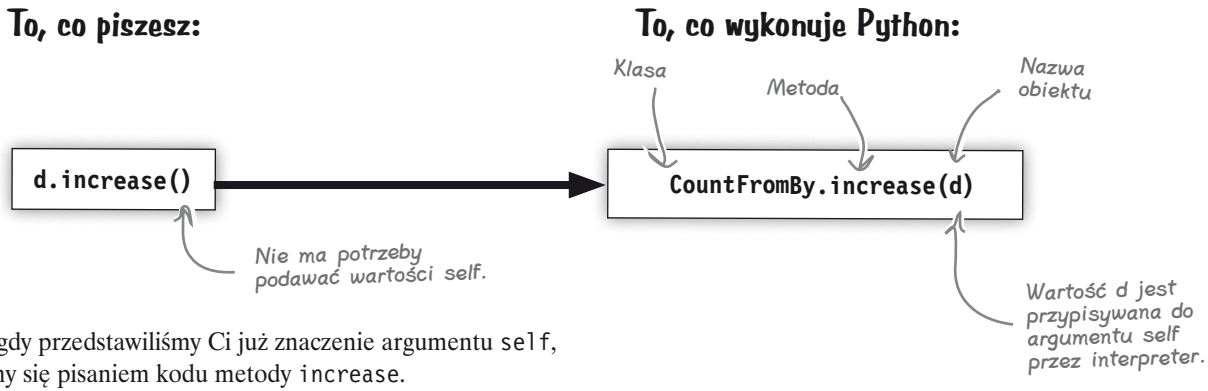
Jednak definiowanie metody bez obowiązkowego pierwszego argumentu spowoduje zgłoszenie przez interpreter błędu przy pierwszym uruchomieniu Twojego kodu. W związku z tym wiersz `def` metody `increase()` w rzeczywistości musi zostać napisany w sposób przedstawiony poniżej.

```
def increase(self):
```

Pisząc kod klasy, traktuj nazwę self jako synonim bieżącego obiektu.

Używanie w kodzie klasy czegoś innego niż nazwa `self` uważane jest za przejaw **bardzo złego stylu**, chociaż do korzystania z tej nazwy trzeba się przyzwyczaić. (W wielu innych językach programowania stosowana jest podobna notacja, przy czym w ich przypadku używa się zwykle nazwy `this`. Koncepcja `self` w języku Python jest zasadniczo tożsama z ideą `this`).

Gdy wywołujesz metodę na rzecz obiektu, Python sprawia, że pierwszym argumentem jest wywołująca instancja, która *zawsze* jest przypisywana do argumentu `self`. Już samo to wyjaśnia, dlaczego jest on tak ważny i dlaczego musi być *pierwszym argumentem* każdej tworzonej przez Ciebie metody obiektu. Gdy wywołujesz metodę, nie musisz podawać wartości `self`, ponieważ interpreter robi to za Ciebie.



Teraz, gdy przedstawiliśmy Ci już znaczenie argumentu `self`, zajmijmy się pisaniem kodu metody `increase`.

Dodawanie metody do klasy

Utwórzmy nowy kod, aby zapisać w nim kod naszej klasy. Nadaj mu nazwę *countfromby.py*, a następnie dodaj do niego kod klasy przedstawiony we wcześniejszej części tego rozdziału i pokazany niżej.

```
class CountFromBy:
    pass
```

Mamy zamiar dodać do tej klasy metodę `increase` i aby to zrobić, usuniemy instrukcję `pass` i zastąpimy ją definicją tej metody. Zanim się tym zajmiemy, przypomnij sobie sposób, w jaki wywoływana jest metoda `increase`.

```
c.increase()
```

W oparciu o to wywołanie można Ci wybaczyć założenie, że metoda `increase` nie przyjmuje żadnych argumentów, ponieważ w nawiasach nic się nie znajduje, prawda? Jest to jednak tylko połowa prawdy. Jak już wiesz, interpreter przetwarza powyższy wiersz kodu w wywołanie przedstawione poniżej.

```
CountFromBy.increase(c)
```

Kod metody, którą piszemy, musi uwzględniać to przekształcenie. Mając to wszystko na uwadze, przyjrzyj się zaprezentowanemu poniżej wierszowi `def` metody `increase`, którą dodamy do naszej klasy.

```
class CountFromBy:
    def increase(self) -> None:
```

Podobnie jak w przypadku innych funkcji w tej książce, dodajemy tu adnotację opisującą typ wartości zwracanej przez metodę.

Metody są jak funkcje, dlatego definiuje się je za pomocą słowa kluczowego def.

Pierwszym argumentem każdej metody jest zawsze self, a jego wartość jest zapewniana przez interpreter.

Metoda `increase` nie przyjmuje już więcej argumentów, dlatego w wierszu `def` nie musimy dodawać już nic więcej niż `self`. *Jest* jednak niezmiernie istotne, aby słowo to znalazło się w definicji funkcji, ponieważ pominięcie go spowoduje wystąpienie błędów składni.

Gdy mamy już napisany wiersz `def`, musimy tylko dodać jakiś kod do metody `increase`. Załóżmy, że klasa ma dwa atrybuty: `val`, zawierający aktualną wartość bieżącego obiektu, oraz `incr`, zawierający liczbę, o którą należy zwiększyć wartość `val` przy każdym wywołaniu metody `increase`. Wiedząc to, możesz poczuć pokusę dodania przedstawionego poniżej **niepoprawnego** wiersza kodu do metody w celu przeprowadzenia odpowiedniej operacji.

```
val += incr
```

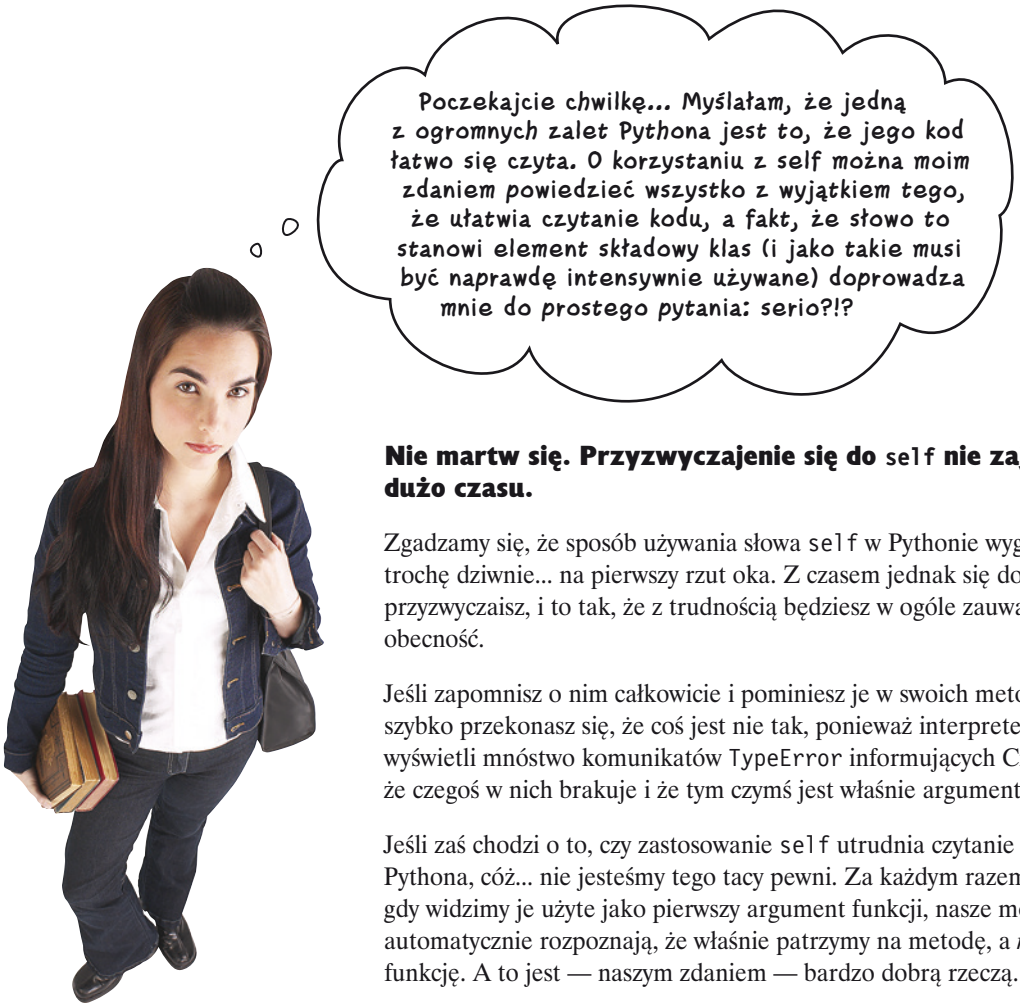
Poniżej znajdziesz jednak **poprawny** wiersz kodu, który powinieneś dodać do metody `increase`.

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

Weź aktualną wartość val przechowywaną w obiekcie i zwiększ ją o liczbę zapisaną w incr.

Jak sądzisz, dlaczego ostatni z przedstawionych wierszy kodu jest prawidłowy, podczas gdy pierwszy taki nie był?

Mówicie poważnie o self?



Nie martw się. Przyzwyczajenie się do `self` nie zajmuje dużo czasu.

Zgadzamy się, że sposób używania słowa `self` w Pythonie wygląda trochę dziwnie... na pierwszy rzut oka. Z czasem jednak się do niego przyzwyczaisz, i to tak, że z trudnością będziesz w ogóle zauważał jego obecność.

Jeśli zapomnisz o nim całkowicie i pominiesz je w swoich metodach, szybko przekonasz się, że coś jest nie tak, ponieważ interpreter wyświetli mnóstwo komunikatów `TypeError` informujących Cię, że czegoś w nich brakuje i że tym czymś jest właśnie argument `self`.

Jeśli zaś chodzi o to, czy zastosowanie `self` utrudnia czytanie kodu Pythona, cóż... nie jesteśmy tego tacy pewni. Za każdym razem, gdy widzimy je użyte jako pierwszy argument funkcji, nasze mózgi automatycznie rozpoznają, że właśnie patrzymy na metodę, a *nie* na funkcję. A to jest — naszym zdaniem — bardzo dobrą rzeczą.

Pomyśl o tym w ten sposób: użycie słowa `self` wskazuje, że kod, który czytasz, stanowi definicję metody, a nie funkcji (w przypadku której słowa tego się *nie* używa).

Duże znaczenie self

W przedstawionym poniżej zestawie metody `increase` każdy z atrybutów klasy poprzedzony jest słowem kluczowym `self`. Poprosiliśmy Cię o przemyślenie, dlaczego musi tak być.

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

O co chodzi z tym słowem self w zestawie metody?

Wiesz już, że interpreter przypisuje zmiennej `self` bieżący obiekt, gdy metoda zostaje wywołana, i że interpreter oczekuje, że uwzględniłeś to, definiując pierwszy argument każdej metody (dzięki czemu przypisanie to może mieć miejsce).

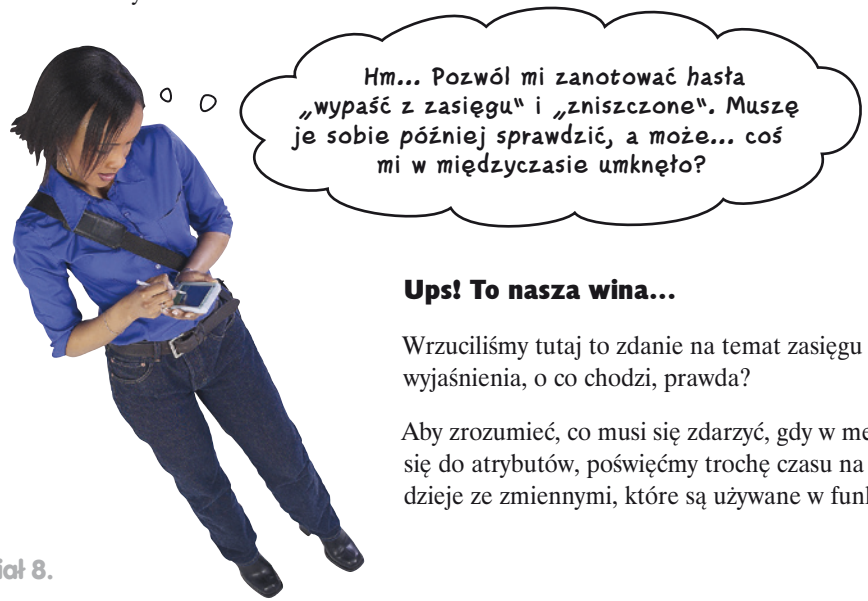
Weź teraz pod uwagę, co wiemy już o każdym istniejącym obiekcie klasy, a mianowicie to, że współdzieli on kod metod klasy (czyli zachowanie) ze wszystkimi innymi obiektami tej samej klasy, ale posiada swoją *własną kopię* wszystkich danych atrybutów (czyli stanu). Odbywa się to w ten sposób, że wartości atrybutów są powiązane z obiektem, a więc ze zmienną `self`.

Mając tę świadomość, przeanalizuj przedstawioną poniżej wersję metody `increase`, o której kilka stron wcześniej napisaliśmy, że jest **nieprawidłowa**.

```
class CountFromBy:
    def increase(self) -> None:
        val += incr
```

Nie rób tak — rozwiązanie to nie będzie działać zgodnie z Twoimi oczekiwaniami.

Na pierwszy rzut oka ostatni spośród wierszy kodu przedstawionych powyżej wygląda całkiem niewinnie, jako że zwiększa jedynie bieżącą wartość zmiennej `val` o liczbę przechowywaną w zmiennej `incr`. Zastanów się jednak, co się dzieje, gdy powyższa metoda `increase` kończy swoje działanie: zmienne `val` oraz `incr`, które istnieją wyłącznie *w obrębie* tej metody, wypadają z zasięgu, w wyniku czego zostają zniszczone w chwili zakończenia metody `increase`.



Ups! To nasza wina...

Wrzuciliśmy tutaj to zdanie na temat zasięgu bez specjalnego wyjaśnienia, o co chodzi, prawda?

Aby zrozumieć, co musi się zdarzyć, gdy w metodzie odwołujesz się do atrybutów, poświęćmy trochę czasu na wyjaśnienie, co się dzieje ze zmiennymi, które są używane w funkcji.

Radzenie sobie z zasięgiem

Aby zademonstrować, co się dzieje ze zmiennymi używanymi w funkcji, poeksperymentujmy trochę w wierszu poleceń powłoki Pythona. Wypróbuj kod przedstawiony poniżej, czytając go. Nasze komentarze ponumerowaliśmy od 1 do 8, aby przeprowadzić Cię przez kolejne kroki przykładu.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> def soundbite(from_outside):
        insider = 'James'
        outsider = from_outside
        print(from_outside, insider, outsider)


>>> name = 'Bond'
>>> soundbite(name)
Bond James Bond
>>> name
'Bond'
>>> insider
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    insider
NameError: name 'insider' is not defined
>>> outsider
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    outsider
NameError: name 'outsider' is not defined
>>> from_outside
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    from_outside
NameError: name 'from_outside' is not defined
>>>
>>>
```

Ln: 42 Col: 4

1. Funkcja soundbite przyjmuje jeden argument.

2. Wartość zostaje przypisana do zmiennej wewnątrz tej funkcji.

3. Argument zostaje przypisany do innej zmiennej wewnątrz tej funkcji.

4. Zmienne funkcji są używane do wyświetlenia komunikatu.

5. Wartość zostaje przypisana do zmiennej o nazwie „name".

6. Funkcja soundbite zostaje wywołana.

7. Po wyświetleniu danych przez funkcję, wartość zmiennej name jest nadal dostępna.

8. Jednak żadna ze zmiennych wykorzystywanych wewnątrz funkcji już nie jest dostępna, ponieważ istnieją one jedynie w obrębie jej zestawu.

Gdy zmienne są definiowane wewnątrz zestawu funkcji, istnieją tylko wtedy, gdy ta funkcja działa. Oznacza to, że zmienne te znajdują się „w zasięgu", dzięki czemu są widoczne i można z nich korzystać w zestawie funkcji. Jednak gdy funkcja się kończy, wszystkie zdefiniowane w jej obrębie zmienne są niszczone — zaczynają znajdować się „poza zasięgiem" — i wszystkie używane przez nie zasoby zostają zwolnione przez interpreter.

To właśnie dzieje się z trzema zmiennymi wykorzystywanymi wewnątrz funkcji `soundbite`, jak widać to powyżej. W chwili gdy funkcja ta dobiega końca, zmienne `insider`, `outsider` i `from_outside` przestają istnieć. Wszelkie próby odwołania się do nich spoza zestawu funkcji (czyli spoza zasięgu tej funkcji) powodują wystąpienie błędu `NameError`.

Poprzedzaj swoje atrybuty nazwą self

Opisany na poprzedniej stronie sposób działania funkcji jest zupełnie w porządku, gdy masz do czynienia z funkcją, która zostaje wywołana, wykonuje jakieś zadanie, a następnie zwraca wartość. Przeważnie nie przejmujesz się w takim przypadku tym, co się dzieje ze zmiennymi, które są używane wewnątrz funkcji, ponieważ interesuje Cię zwykle jedynie wartość zwracana przez funkcję.

Teraz, skoro wiesz już, co się dzieje ze zmiennymi, gdy funkcja kończy swoje działanie, powinno już być dla Ciebie jasne, że przedstawiony poniżej (niepoprawny) kod prawdopodobnie spowoduje problemy, jeśli spróbujesz wykorzystać zmienne do przechowywania i zapamiętywania wartości atrybutów w klasie. Jako że metody to noszące inną nazwę funkcje, jeśli metodę `increase` zaimplementujesz w sposób zaprezentowany poniżej, zmienne `val` i `incr` zostaną zniszczone, gdy tylko zakończy się działanie tej metody.

```
class CountFromBy:
    def increase(self) -> None:
        val += incr
```

Nie rób tak, ponieważ te zmienne nie przetrwają, gdy metoda zakończy swoje działanie.

W przypadku metod sprawy mają się jednak *inaczej*. W metodzie używa się wartości atrybutów, które należą do obiektu, a atrybuty te istnieją nadal *po* tym, jak metoda zakończy swoje działanie. Oznacza to, że wartości atrybutów obiektów **nie** są niszczone, gdy metoda się kończy.

Jeśli chcesz, aby wartość atrybutu przetrwała zakończenie działania metody, wartość tę powinieneś przypisać do czegoś, co nie zostaje zniszczone w wyniku tego zakończenia. Tym *czymś* jest bieżący obiekt, na rzecz którego metoda jest wywoływana, a więc obiekt przechowywany w zmiennej `self`, co tłumaczy, dlaczego każdą wartość atrybutu należy poprzedzać tą nazwą w kodzie swoich metod, tak jak zostało to pokazane poniżej.

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

To znacznie lepsze rozwiązanie, ponieważ dzięki zastosowaniu słowa self zmienne val i incr są teraz powiązane z naszym obiektem.

Nazwa „self" stanowi synonim bieżącego obiektu.

Zasada jest tu prosta: jeśli w swojej klasie chcesz odwołać się do atrybutu, *musisz* poprzedzić jego nazwę słowem `self`. Wartość reprezentowana przez to słowo stanowi *synonim* wskazujący obiekt, na rzecz którego wywoływana jest metoda.

W tym kontekście, gdy widzisz słowo `self`, pomyśl „ten obiekt". Zatem zapis `self.val` powinieneś odczytywać jako „atrybut `val` należący do tego obiektu".

Inicjalizuj wartości (atrybutów) przed użyciem

Cała ta dyskusja na temat znaczenia słowa `self` zepchnęła na bok pewną istotną kwestię dotyczącą tego, w jaki sposób atrybutom przypisuje się wartości początkowe. Próba wykonania metody `increase` w takiej postaci, jaką ma ona obecnie, zakończy się niepowodzeniem, mimo że mamy tu na myśli prawidłową wersję kodu czyli tę, w której zastosowaliśmy słowo `self`. Stanie się tak, ponieważ nie można użyć zmiennej, zanim nie zostanie jej przypisana wartość, niezależnie od tego, gdzie zmienna ta jest wykorzystywana.

Aby przekonać się o powadze sytuacji, przyjrzyj się przedstawionemu poniżej zapisowi sesji w powłoce Pythona. Zauważ, że próba wykonania pierwszej instrukcji kończy się niepowodzeniem, gdy *którakolwiek* z zastosowanych zmiennych jest niezdefiniowana.

Gdy próbujesz wykonać kod odwołujący się do niezainicjalizowanych zmiennych...

```
>>> val += incr
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    val += incr
NameError: name 'val' is not defined
```

...interpreter „narzeka".

Interpreter odmawia uruchomienia tego wiersza kodu, ponieważ zmienna val jest niezdefiniowana.

Przypisz wartość do zmiennej val, a następnie spróbuj raz jeszcze...

```
>>> val = 0

>>> val += incr
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    val += incr
NameError: name 'incr' is not defined
```

...interpreter znowu „narzeka"!

Interpreter nadal odmawia uruchomienia tego wiersza kodu, ponieważ zmienna incr jest niezdefiniowana.

Przypisz wartość do zmiennej incr i spróbuj ponownie...

```
>>> incr = 1

>>> val += incr
```

...tym razem się udało.

```
>>> val
1
>>> incr
1
>>>
```

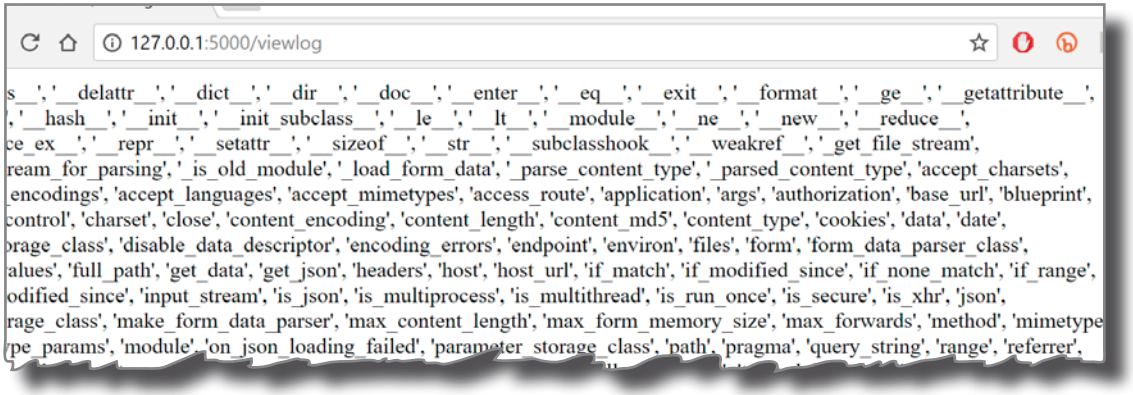
Interpreter może użyć wartości zmiennych val oraz incr bez zgłaszania błędu NameError, ponieważ obydwie te wartości zostały zdefiniowane (a więc zmienne są zainicjalizowane).

W języku Python niezależnie od tego, gdzie używasz zmiennych, musisz je zainicjalizować jakimiś wartościami początkowymi. Rodzi się tu pytanie: *jak to zrobić w przypadku nowo utworzonego obiektu klasy Pythona?*

Jeśli znasz paradygmat OOP, w Twoim mózgu może się w tym momencie pojawić słowo „konstruktor". W innych językach programowania konstruktor jest specjalną metodą, która umożliwia Ci definiowanie, co się dzieje, gdy obiekt jest tworzony, a zwykle wiąże się to zarówno z tworzeniem samej instancji, jak i inicjalizacją jej atrybutów. W Pythonie tworzenie instancji odbywa się automatycznie dzięki interpreterowi, dlatego nie musisz w tym celu definiować konstruktora. Zgodną z Twoimi potrzebami inicjalizację obiektów umożliwi Ci z kolei metoda o nazwie `__init__`. Przekonajmy się, co ten dunder `init` jest w stanie nam zaoferować.

Dunder init umożliwia inicjalizowanie atrybutów

Wróć pamięcią do jednego z ostatnich rozdziałów, w którym korzystałeś z wbudowanej funkcji `dir`, aby wyświetlić wszystkie szczegóły dotyczące obiektu `req` frameworka Flask. Pamiętasz, co udało Ci się wtedy uzyskać?



Spójrz tylko na wszystkie te dundery!

Gdy prezentowaliśmy Ci te dane, sugerowaliśmy, abyś zignorował wszystkie te dundery. Teraz nadszedł jednak czas na ujawnienie ich przeznaczenia: widoczne tu dundery mają związek z każdym standardowym zachowaniem klasy.

Jeśli nie zadbasz o ich przesłonięcie, zachowania te będą implementowane w klasie o nazwie `object`. Klasa ta jest wbudowana w interpreter i wszystkie inne klasy Pythona (włącznie z Twoimi) *automatycznie* po niej dziedziczą. W żargonie obiektowym oznacza to, że metody dunder zapewniane przez klasę `object` są dostępne w Twoich klasach i możesz ich używać lub też możesz je przesłonić, jeśli tylko zachodzi taka potrzeba (zapewniając własną implementację tych metod).

Nie musisz przesłaniać żadnych metod klasy `object`, jeśli nie chcesz tego robić. Jeżeli jednak zamierzasz na przykład określić, co się dzieje, gdy obiekt Twojej klasy jest używany wraz z operatorem równości (`==`), wówczas możesz napisać własny kod metody `__eq__`. Jeśli chcesz zdefiniować, co ma się stać, gdy obiekt będzie używany w połączeniu z operatorem większości (`>`), możesz przesłonić metodę `__ge__`. A jeśli masz zamiar *zainicjalizować* atrybuty związane z Twoim obiektem, możesz skorzystać w tym celu z metody `__init__`.

Standardowe metody dunder udostępniane przez wszystkie klasy są znane jako magiczne metody.

Z uwagi na to, że dundery zapewniane przez klasę `object` są tak bardzo przydatne, są one otoczone niemal mistyczną czcią wśród programistów Pythona. Uczucie to jest tak silne, że wielu z nich w rzeczywistości określa je mianem *magicznych metod* (ponieważ wydają się spełniać swoje zadania w sposób wręcz magiczny).

Wszystko to oznacza, że jeśli w swojej klasie zdefiniujesz metodę `__init__` za pomocą wiersza `def` takiego, jaki został pokazany poniżej, interpreter wywoła ją za każdym razem, gdy będziesz tworzył nowy obiekt tej klasy. Zwróć uwagę na fakt, że pierwszym argumentem tej metody `__init__` jest `self` (zgodnie z regułą dotyczącą wszystkich metod wszystkich klas).

```
def __init__(self):
```

Mimo dziwnie wyglądającej nazwy dunder init jest taką samą metodą jak wszystkie inne. Pamiętaj, że również w jej przypadku pierwszym argumentem musi być self.

Inicjalizowanie atrybutów za pomocą dundera init

Dodajmy metodę `__init__` do naszej klasy `CountFromBy` w celu zapewnienia inicjalizacji obiektów tej klasy, które tworzymy.

Na razie dodajmy jedynie *pustą* metodę `__init__`, która nie wykonuje żadnej operacji oprócz instrukcji `pass` (definiowaniem jej prawdziwego zachowania zajmiemy się za chwilę).

```
class CountFromBy:
    def __init__(self) -> None:
        pass
    def increase(self) -> None:
        self.val += self.incr
```

W tym momencie metoda __init__ nie robi zupełnie nic. Jednak użycie self w roli pierwszego argumentu stanowi WYRAŹNY SYGNAŁ, że dunder init to metoda.

Dzięki kodowi metody `increase` wiemy już, że do atrybutów naszej klasy możemy uzyskać dostęp, korzystając z przedrostka `self`. Oznacza to, że możemy użyć konstrukcji `self.val` oraz `self.incr`, aby odwołać się do naszych atrybutów również z wnętrza metody `__init__`. Chcemy jednak skorzystać z tej metody, aby *zainicjalizować* te atrybuty (`val` oraz `incr`). Rodzi się tu pytanie: skąd wziąć wartości początkowe i jak przekazać je do metody `__init__`?

Przekaż dowolną liczbę argumentów do dundera init

Jako że `__init__` jest metodą, a metody to funkcje w przebraniu, możesz przekazać jej dowolną liczbę argumentów (podobnie zresztą jak każdej innej metodzie). Musisz jedynie nadać tym argumentom nazwy. Argumentowi, z którego wartości skorzystamy, aby zainicjalizować atrybut `self.val`, nadajmy nazwę `v`, a argument odpowiedzialny za inicjalizację atrybutu `self.incr` nazwijmy `i`.

Dodajmy te argumenty do wiersza `def` metody `__init__`, a następnie użyjmy ich wartości w jej zestawie, aby zainicjalizować odpowiednie atrybuty klasy.

```
class CountFromBy:
    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i
    def increase(self) -> None:
        self.val += self.incr
```

Dodaj argumenty v oraz i do definicji metody __init__.

Używa wartości v oraz i do inicjalizacji atrybutów klasy (którymi są, odpowiednio, self.val oraz self.incr).

Gdybyśmy byli teraz w stanie przekazać wartości argumentom `v` oraz `i`, to za pomocą ostatniej wersji metody `__init__` moglibyśmy zainicjalizować atrybuty naszej klasy. Prowadzi to do kolejnego pytania: jak mamy dostarczyć te wartości? Aby odpowiedzieć na to pytanie, powinniśmy wypróbować tę wersję naszej klasy i przekonać się, jak działa. Zajmijmy się tym teraz.

Jazda próbna

Poświęć trochę czasu, aby korzystając z okna edycji środowiska IDLE, zaktualizować kod znajdujący się w pliku *countfromby.py* do postaci przedstawionej poniżej. Po zrobieniu tego naciśnij klawisz *F5*, aby zacząć tworzyć obiekty w wierszu poleceń IDLE.

Naciśnij klawisz F5, aby wypróbować klasę CountFromBy w powłoce IDLE.

countfromby.py - C:/Users/lsuma/Desktop/Python/R08...

File Edit Format Run Options Window Help

```
class CountFromBy:
    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr
```

Ln: 8 Col: 0

To jest najnowsza wersja naszej klasy CountFromBy.

Naciśnięcie klawisza *F5* powoduje wykonanie kodu widocznego w oknie edycji, w wyniku czego klasa `CountFromBy` importowana jest do interpretera. Poniżej możesz zobaczyć, co się stało, gdy spróbowaliśmy utworzyć nowy obiekt tej klasy.

Utwórz nowy obiekt klasy (o nazwie „g"), ale... gdy próbujesz to zrobić, na ekranie pojawia się komunikat błędu!

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: C:/Users/lsuma/Desktop/Python/R08/countfromby.py =========
>>>
>>> g = CountFromBy()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    g = CountFromBy()
TypeError: __init__() missing 2 required positional arguments: 'v' and 'i'
>>>
>>>
```

Ln: 12 Col: 4

Prawdopodobnie nie to spodziewałeś się ujrzeć. Przyjrzyj się jednak wyświetlonemu w oknie IDLE komunikatowi błędu (który został zakwalifikowany jako `TypeError`), zwracając szczególną uwagę na wiadomość widoczną w wierszu zawierającym słowo `TypeError`. Interpreter informuje nas tutaj, że metoda `__init__` spodziewała się otrzymać dwie wartości argumentów, `v` oraz `i`, ale otrzymała coś innego (w naszym przypadku nic). Nie dostarczyliśmy w tym miejscu żadnych wartości, ale ten komunikat błędu informuje nas, że wszystkie argumenty podane klasie (podczas tworzenia nowego obiektu) są przekazywane metodzie `__init__`.

Mając to na uwadze, podejmijmy kolejną próbę utworzenia obiektu klasy `CountFromBy`.

Powróćmy do okna wiersza poleceń powłoki Pythona i utwórzmy następny obiekt (o nazwie h), który przyjmuje dwie wartości (dla dwóch argumentów: v oraz i).

Tym razem brak jest błędu TypeError.

Możesz odwołać się do wartości atrybutów obiektu h.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> h = CountFromBy(100, 10)
>>> h.val
100
>>> h.incr
10
>>> h.increase()
>>> h.val
110
>>> h
<__main__.CountFromBy object at 0x00000211E976FEF0>
>>>
>>>
Ln: 23  Col: 4
```

Wywołanie metody increase odnosi dokładnie taki skutek, jakiego się spodziewasz. Powoduje zwiększenie atrybutu h.val o wartość przechowywaną w atrybucie h.incr.

Prawdopodobnie spodziewałeś się tu zobaczyć wartość 110, ale zamiast tego ujrzałeś ten (dość tajemniczy) komunikat.

Jak możesz się przekonać powyżej, sprawy mają się tym razem znacznie lepiej, ponieważ nie występuje już wyjątek `TypeError`, co oznacza, że udało się utworzyć obiekt h. Możesz odwołać się do wartości atrybutów tego obiektu, korzystając z konstrukcji `self.val` oraz `self.incr`, a także wywoływać na jego rzecz metodę `increase`. Dziwne rzeczy dzieją się tylko, gdy próbujesz odwołać się do wartości samego h.

Czego nauczyliśmy się z tej jazdy próbnej?

To są najważniejsze wnioski, które wyciągnęliśmy z tej jazdy próbnej:

- Gdy tworzysz obiekty, wszystkie wartości argumentów podane klasie są przekazywane jej metodzie `__init__`, tak jak miało to miejsce w przypadku występujących powyżej wartości 100 oraz 10. (Zwróć uwagę, że zmienne v oraz i przestają istnieć, gdy tylko metoda `__init__` kończy swoje działanie, ale w ogóle nas to nie martwi, jako że ich wartości zostały bezpiecznie zapisane w atrybutach należących do obiektu, odpowiednio, `self.val` i `self.incr`).
- Do wartości atrybutów można się odwołać, łącząc za pomocą kropki nazwę obiektu z nazwą odpowiedniego atrybutu. Zauważ, jak korzystaliśmy z tej możliwości w przypadku atrybutów `self.val` i `self.incr`. (Osoby, które przechodzą do Pythona z innych, bardziej „rygorystycznych" języków zorientowanych obiektowo, powinny zwrócić uwagę, że mogliśmy to zrobić bez konieczności wcześniejszego definiowania odpowiednich *getterów* i *setterów*).
- Gdy korzystamy z samej nazwy obiektu (jak ma to miejsce w przypadku ostatniej instrukcji pokazanej powyżej w wierszu poleceń), interpreter „wypluwa" dość tajemniczy komunikat. Co on zawiera (i dlaczego tak wygląda), omówimy już za chwilę.

Znaczenie reprezentacji klasy CountFromBy

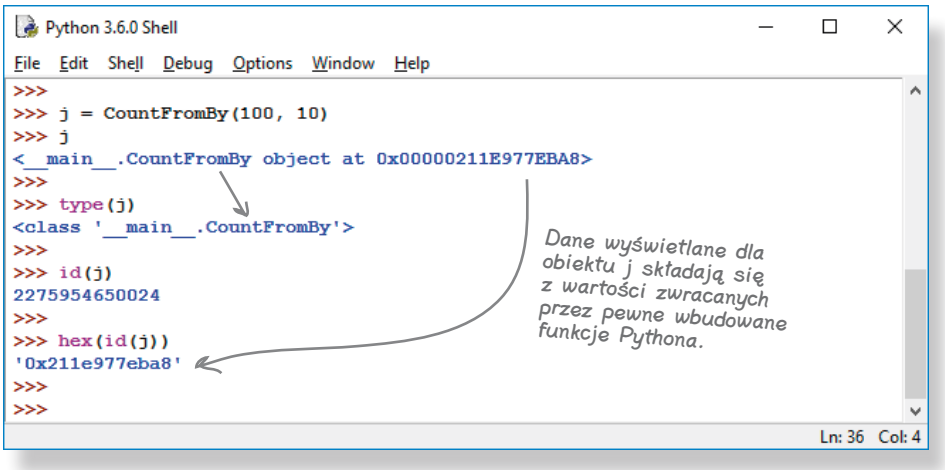
Gdy wpisaliśmy nazwę obiektu w powłoce Pythona w celu wyświetlenia jego bieżącej wartości, interpreter wyświetlił nam dane przedstawione poniżej.

Nie martw się, jeśli na swoim komputerze otrzymujesz tu inną wartość. Wszystko się wyjaśni przed końcem tej strony.

```
<__main__.CountFromBy object at 0x00000211E976FEF0>
```

Opisując to, co widzisz powyżej, użyliśmy słowa „dziwne” i na pierwszy rzut oka właśnie takie się to wydaje. Aby zrozumieć, co oznaczają te dane, powróćmy do powłoki IDLE i utwórzmy kolejny obiekt klasy `CountFromBy`, który w związku z naszą głęboko zakorzenioną zachowawczością nazwiemy `j`.

Spójrz na przedstawioną poniżej sesję i zwróć uwagę, że dziwny komunikat wyświetlony dla obiektu `j` składa się z wartości, które otrzymaliśmy, wywołując określone wbudowane funkcje Pythona. Spróbuj najpierw powtórzyć nasze kroki na swoim komputerze, a dopiero potem przeczytaj znajdujące się pod spodem wyjaśnienia dotyczące działań wykonywanych przez te funkcje.



Wbudowana funkcja `type` wyświetla informację na temat klasy obiektu, powiadamiając nas (powyżej), że `j` jest obiektem klasy `CountFromBy`.

Wbudowana funkcja `id` wyświetla informację na temat adresu, pod którym obiekt znajduje się w pamięci (będącego unikatowym identyfikatorem używanym przez interpreter w celu śledzenia Twoich obiektów). To, co możesz teraz zaobserwować na swoim ekranie, prawdopodobnie różni się od tego, co widać powyżej.

Adres obiektu w pamięci wyświetlony jako element składowy danych obiektu `j` stanowi wartość `id` skonwertowaną na postać szesnastkową (konwersję tę można przeprowadzić za pomocą wbudowanej funkcji `hex`). A zatem cały komunikat wyświetlony dla obiektu `j` jest kombinacją jego typu i identyfikatora (skonwertowanego na postać heksadecymalną).

Warto zadać sobie tu pytanie: *dlaczego tak się dzieje?*

Jeśli nie poinformujesz interpretera, jaka ma być reprezentacja Twoich obiektów, w odpowiedzi na żądanie wyświetlenia obiektu musi on *coś* zrobić, więc robi to, co zostało przedstawione powyżej. Na szczęście możesz przesłonić to standardowe zachowanie, tworząc własną wersję magicznej metody `__repr__`.

Przesłoń dunder repr, aby określić sposób, w jaki Twoje obiekty mają być reprezentowane przez interpreter.

Definiowanie reprezentacji klasy CountFromBy

Zadanie spełniane przez magiczną metodę `__repr__` jest również realizowane przez wbudowaną funkcję o nazwie `repr`. Oto co na jej temat ma do „powiedzenia" wbudowana funkcja `help`: „Zwraca reprezentację obiektu w postaci kanonicznego łańcucha znakowego". Innymi słowy, pomoc Pythona informuje Cię, że funkcja `repr` (a co za tym idzie, także metoda `__repr__`) powinna zwracać tekstową wersję obiektu.

Dokładny wygląd „tekstowej wersji obiektu" zależy od tego, co robi konkretny obiekt. Możesz kontrolować, co dzieje się tu w przypadku *Twoich* obiektów, pisząc dla swojej klasy własny kod metody `__repr__`. Zróbmy to dla klasy `CountFromBy`.

Zacznij od dodania do klasy `CountFromBy` nowego wiersza `def` dla dundera `repr`, który nie przyjmuje żadnych argumentów oprócz obowiązkowego `self` (pamiętaj, że masz tu do czynienia z metodą). Zgodnie z naszą praktyką dodajmy tu również adnotację, która poinformuje czytelników naszego kodu, że metoda ta zwraca łańcuch znakowy.

Podobnie jak wszystkie inne metody, które piszesz, również ta musi brać pod uwagę, że interpreter zawsze zapewnia wartość pierwszego argumentu.

```
def __repr__(self) -> str:
```

To informuje użytkowników metody, że ma ona zwracać łańcuch znakowy. Pamiętaj, że stosowanie w Twoim kodzie adnotacji jest opcjonalne, ale bardzo przydatne.

Po napisaniu wiersza `def` zostanie nam do opracowania już tylko kod, który zwraca tekstową reprezentację obiektu klasy `CountFromBy`. W naszym przypadku reprezentacją tą powinna być skonwertowana na postać łańcucha znakowego wartość atrybutu `self.val`.

Dzięki wbudowanej funkcji `str` zadanie to jest bardzo proste.

```
def __repr__(self) -> str:
    return str(self.val)
```

Weź wartość zapisaną w atrybucie self.val, przekształć ją w łańcuch znakowy, a następnie zwróć do kodu wywołującego tę metodę.

Jeśli dodasz tę krótką funkcję do swojej klasy, interpreter będzie z niej korzystał zawsze wtedy, gdy zajdzie konieczność wyświetlenia obiektu klasy `CountFromBy` w wierszu poleceń powłoki Pythona. Wbudowana funkcja `print` również korzysta z dundera `repr` przy wyświetlaniu obiektów.

Zanim wprowadzisz tę zmianę w swoim kodzie i weźmiesz jego zaktualizowaną wersję w obroty, wróćmy jeszcze na krótką chwilę do innego problemu, który wyłonił się podczas ostatniej jazdy próbnej.

Zapewnianie sensownych wartości domyślnych dla klasy CountFromBy

Przypomnijmy sobie, jak wygląda bieżąca wersja metody `__init__` klasy `CountFromBy`.

```
    ...
def __init__(self, v: int, i: int) -> None:
    self.val = v
    self.incr = i
    ...
```

Ta wersja metody dunder init „spodziewa się" dwóch wartości argumentów podanych za każdym razem, gdy jest ona wywoływana.

Przypomnij sobie, że gdy próbowaliśmy utworzyć nowy obiekt tej klasy, nie podając wartości argumentów `v` oraz `i`, otrzymaliśmy błąd `TypeError`, co widać poniżej.

```
===== RESTART: C:/Users/lsuma/Desktop/Python/R08/countfromby.py ========

g = CountFromBy()
eback (most recent call last):
le "<pyshell#1>", line 1, in <module>
g = CountFromBy()
Error: __init__() missing 2 required positional arguments: 'v' and 'i'
```

Oj! Niedobrze.

We wcześniejszej części tego rozdziału wspomnieliśmy, że chcemy, aby klasa `CountFromBy` zapewniała następujące zachowanie standardowe: licznik rozpoczyna swoją pracę od wartości 0 i (na życzenie) jest zwiększany o 1. Wiesz już, jak zapewnić argumentom funkcji wartości domyślne, a tę samą technikę można stosować również w przypadku metod; odpowiednie wartości trzeba po prostu podać w wierszu `def`.

```
    ...
def __init__(self, v: int=0, i: int=1) -> None:
    self.val = v
    self.incr = i
    ...
```

Z racji tego, że metody są funkcjami, umożliwiają korzystanie z wartości domyślnych argumentów (w naszym przypadku jest to początkowa wartość licznika dostarczona argumentowi `v` oraz wartość, o którą domyślnie należy go zwiększać, podana argumentowi `i`).

Po wprowadzeniu tej niewielkiej (lecz istotnej) zmiany w kodzie swoje klasy `CountFromBy` zapisz przechowujący ją plik (zanim ponownie naciśniesz klawisz *F5*), a przekonasz się, że obiekty można teraz tworzyć, korzystając z tego standardowego zachowania.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> i
>>> i = CountFromBy()
>>> i.val
0
>>> i.incr
1
>>> i.increase()
>>> i.val
1
>>>
>>>
```

Nie określiliśmy wartości, które mają zostać użyte do inicjalizacji obiektu, dlatego klasa zapewniła wartości domyślne, jakie zostały zdefiniowane w dunderze init.

Wszystko działa tu zgodnie z oczekiwaniami; metoda increase zwiększa wartość atrybutu i.val o 1 za każdym razem, gdy zostaje wywołana. Jest to standardowe zachowanie klasy.

Jazda próbna

Upewnij się, że kod Twojej klasy `CountFromBy` (znajdujący się w pliku *countfromby.py*) jest dokładnie taki sam, jak przedstawiony poniżej nasz kod. Przejdź do okna edycji środowiska IDLE, załaduj w nim ten kod i naciśnij klawisz *F5*, aby wziąć swoją klasę w obroty.

To jest kod klasy CountFromBy z dodanym kodem dundera repr.

countfromby.py - C:/Users/lsuma/Desktop/Python/R08...

File Edit Format Run Options Window Help

```
class CountFromBy:
    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

Ln: 12 Col: 0

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> k = CountFromBy()
>>> k
0
>>> k.increase()
>>> k
1
>>> print(k)
1
>>> l = CountFromBy(100)
>>> l
100
>>> l.increase()
>>> print(l)
101
>>> m = CountFromBy(100, 10)
>>> m
100
>>> m.increase()
>>> m
110
>>> n = CountFromBy(i=15)
>>> n
0
>>> n.increase()
>>> n
15
>>>
```

Ln: 32 Col: 4

Obiekt k korzysta z domyślnych wartości, tzn. zaczyna zliczanie od 0 i jest zwiększany o 1.

Gdy odwołujesz się do obiektu w wierszu poleceń lub w wywołaniu funkcji print, uruchamiany jest kod dundera repr.

Obiekt l zapewnia alternatywną wartość początkową, a następnie zwiększa licznik o 1 przy każdym wywołaniu metody increase.

Obiekt m korzysta z alternatywnych wartości obydwu argumentów.

Obiekt n korzysta ze słowa kluczowego argumentu w celu określenia alternatywnej wartości, o którą należy zwiększać licznik (ale rozpoczyna zliczanie od 0).

Klasy: co już wiemy

Skoro już opracowaliśmy klasę `CountFromBy` działającą w taki sposób, jak zostało to określone na początku tego rozdziału, przejrzyjmy informacje, które zdobyliśmy na temat klas w języku Python.

CELNE SPOSTRZEŻENIA

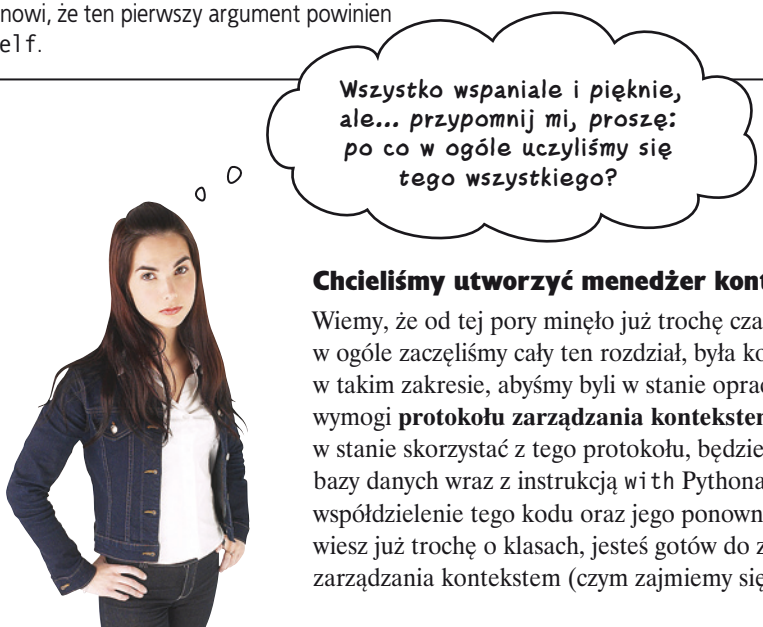
- Klasy Pythona umożliwiają Ci współdzielenie **zachowania** (czyli metod) oraz **stanu** (czyli atrybutów).
- Jeśli zapamiętasz, że metody są **funkcjami**, a atrybuty **zmiennymi**, nie będziesz daleki od prawdy.
- Słowo kluczowe `class` stanowi początek definicji klasy w Twoim kodzie.
- Tworzenie nowego obiektu klasy bardzo przypomina wywołanie funkcji. Pamiętaj, że aby utworzyć obiekt o nazwie `mycount` klasy `CountFromBy`, powinieneś napisać następujący wiersz kodu:

```
mycount = CountFromBy()
```

- Obiekt danej klasy współdzieli jej kod ze wszystkimi innymi obiektami tej klasy, jednak każdy obiekt posiada swoją **własną kopię** atrybutów.
- Klasy wyposaża się w zachowania, tworząc **metody**. Metoda to funkcja zdefiniowana wewnątrz klasy.
- Aby dodać do klasy **atrybut**, tworzy się zmienne.
- Każdej metodzie jako pierwszy argument przekazywany jest **synonim** bieżącego obiektu. Używana w Pythonie konwencja stanowi, że ten pierwszy argument powinien nosić nazwę `self`.
- Odwołania do atrybutów wewnątrz zestawu metody są poprzedzane przedrostkiem `self`; dzięki temu wartości atrybutów są w stanie **przetrwać** zakończenie działania metody.
- Metoda `__init__` jest jedną z **magicznych metod** zapewnianych wszystkim klasom Pythona.
- Wartości atrybutów są inicjalizowane przez metodę `__init__` (znaną również jako dunder `init`). Metoda ta umożliwia przypisywanie wartości początkowych do atrybutów, gdy tworzy się nowe obiekty. Dunder `init` przyjmuje **kopię** wszystkich wartości, które są przekazywane klasie, gdy powstaje instancja. Na przykład wartości 100 i 10 są przekazywane metodzie `__init__`, gdy tworzony jest poniższy obiekt:

```
mycount2 = CountFromBy(100, 10)
```

- Kolejną magiczną metodą jest `__repr__`, która umożliwia Ci kontrolowanie sposobu, w jaki obiekty są wyświetlane w wierszu poleceń powłoki Pythona, jak również wtedy, gdy odwołuje się do nich w wywołaniach funkcji wbudowanych, takich jak `print`.



Chcieliśmy utworzyć menedżer kontekstu.

Wiemy, że od tej pory minęło już trochę czasu, ale powodem, dla którego w ogóle zaczęliśmy cały ten rozdział, była konieczność poznania klas w takim zakresie, abyśmy byli w stanie opracować kod, który spełnia wymogi **protokołu zarządzania kontekstem** języka Python. Jeśli będziemy w stanie skorzystać z tego protokołu, będziemy też mogli użyć kodu obsługi bazy danych wraz z instrukcją `with` Pythona, co znacznie ułatwi nam współdzielenie tego kodu oraz jego ponowne wykorzystanie. Teraz, gdy wiesz już trochę o klasach, jesteś gotów do zmierzenia się z protokołem zarządzania kontekstem (czym zajmiemy się w następnym rozdziale).

Kod przykładów z rozdziału 8.

To jest kod zapisany w pliku countfromby.py.

```
class CountFromBy:
    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

Korzystanie z instrukcji with języka Python

Pora w praktyce skorzystać z tego, czego się właśnie nauczyłeś.

W rozdziale 7. omówiliśmy używanie **relacyjnej bazy danych** w Pythonie, natomiast rozdział 8. stanowił wstęp do korzystania z **klas** w Twoim kodzie napisanym w tym języku. W niniejszym rozdziale obydwie te rzeczy zostaną połączone w celu utworzenia **menedżera kontekstu**, który pozwoli nam rozszerzyć instrukcję `with` w taki sposób, aby mogła współpracować z systemami relacyjnych baz danych. Będziesz tu korzystać z instrukcji `with`, tworząc nową klasę, która spełnia wymagania **protokołu zarządzania kontekstem** języka Python.

W jaki sposób najlepiej współdzielić kod obsługi bazy danych naszej aplikacji WWW?

W rozdziale 7. opracowałeś kod obsługi bazy danych i umieściłeś go w swojej funkcji `log_request`, która działała całkiem dobrze, ale musiałeś wstrzymać się z dalszą pracą, aby zastanowić się nad najlepszym sposobem współdzielenia tego kodu. Przypomnij sobie sugestie zaproponowane pod koniec rozdziału 7., które zostały pokazane poniżej.

Stwierdziliśmy wtedy, że każda z tych sugestii jest na swój sposób słuszna, ale prawdziwi programiści Pythona raczej nie wybraliby żadnego z tych rozwiązań *z osobna*. Zdecydowaliśmy, że lepiej tu będzie skorzystać z protokołu zarządzania kontekstem i możliwości oferowanych przez instrukcję `with`, lecz aby to zrobić, musiałeś dowiedzieć się nieco o klasach. Im poświęcony był ostatni rozdział. Teraz, gdy wiesz już, jak tworzyć klasę, nadszedł czas na powrót do naszego zadania polegającego na tworzeniu menedżera kontekstu, dzięki któremu będziemy w stanie współdzielić kod obsługi bazy danych wykorzystywany w naszej aplikacji WWW.

Ponownie zastanów się, co próbujesz zrobić

Poniżej znajdziesz nasz kod obsługi bazy danych przedstawiony w rozdziale 7. Kod ten stanowi aktualnie część naszej aplikacji korzystającej z frameworka Flask. Przypomnij sobie, jak za pomocą tego kodu łączyła się ona z bazą danych MySQL, zapisywała szczegóły dotyczące żądań sieciowych w tabeli `log`, zatwierdzała wszystkie *niezapisane* dane, a następnie rozłączała się z bazą.

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""

    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))

    conn.commit()
    cursor.close()
    conn.close()
```

Ten słownik zawiera szczegóły związane z właściwościami połączenia z bazą danych.

W tym fragmencie dane ze słownika są wykorzystywane w celu nawiązania połączenia, a następnie tworzony jest kursor.

To jest kod, który odpowiada za faktyczne działanie; jego zadaniem jest dodanie danych żądania do tabeli log bazy danych.

Na zakończenie za pomocą tego kodu zamykane jest połączenie.

Jak najlepiej utworzyć menedżer kontekstu?

Zanim znajdziemy się w punkcie, w którym będziesz w stanie przekształcić pokazany powyżej kod w coś, co można wykorzystać jako część instrukcji `with`, omówmy, jak da się to osiągnąć w zgodzie z protokołem zarządzania kontekstem. Chociaż biblioteka standardowa umożliwia tworzenie prostych menedżerów kontekstu (za pomocą modułu `contextlib`), korzystanie z klasy zgodnej z tym protokołem uznaje się za prawidłowe podejście, gdy za pomocą instrukcji `with` chce się kontrolować jakiś zewnętrzny obiekt, taki jak połączenie z bazą danych (z czym mamy do czynienia w naszym przykładzie).

Mając to na uwadze, przekonajmy się, czym jest owa „zgodność z protokołem zarządzania kontekstem”.

Zarządzanie kontekstem za pomocą metod

Określenie „protokół zarządzania kontekstem” brzmi groźnie i onieśmielająco, w rzeczywistości jednak opisuje dość prostą rzecz. Protokół ten dyktuje, że każda tworzona przez Ciebie klasa musi definiować co najmniej dwie magiczne metody: `__enter__` oraz `__exit__`. To jest protokół. Gdy się do niego stosujesz, Twoja klasa może łączyć się z instrukcją `with`.

Protokół to ustalona procedura (lub zbiór procedur), do których należy się stosować.

Dunder enter odpowiada za konfigurację

Gdy obiekt jest używany wraz z instrukcją `with`, interpreter wywołuje należącą do niego metodę `__enter__`, *zanim* rozpocznie się wykonywanie zestawu tej instrukcji. Daje Ci to możliwość wykonania kodu odpowiedzialnego za wszelkie niezbędne działania konfiguracyjne w obrębie tej metody.

Protokół określa również, że dunder `enter` może (ale nie musi) zwracać wartość do instrukcji `with` (dlaczego jest to istotne, przekonasz się już wkrótce).

Dunder exit odpowiada za demontaż

Gdy tylko kończy się wykonanie zestawu instrukcji `with`, interpreter *zawsze* wywołuje należącą do obiektu metodę `__exit__`. Odbywa się to już po zakończeniu działania kodu tworzącego ten zestaw i jest okazją do przeprowadzenia wszelkich działań związanych ze sprzątaniem.

Jako że kod należący do zestawu instrukcji `with` może w jakiś sposób zawieść (i spowodować wystąpienie wyjątku), dunder `exit` musi być gotów do obsługi sytuacji tego rodzaju. Do problemu tego powrócimy w dalszej części niniejszego rozdziału, gdy będziemy tworzyć kod naszej metody `__exit__`.

Jeśli utworzysz klasę, w której zdefiniowane są metody `__enter__` oraz `__exit__`, klasa ta jest przez interpreter automatycznie uznawana za menedżer kontekstu, dzięki czemu może łączyć się z instrukcją `with` (i być z nią używana). Innymi słowy, klasa taka *jest zgodna* z protokołem zarządzania kontekstem i *implementuje* menedżer kontekstu.

Jeśli Twoja klasa definiuje dundery enter i exit, stanowi menedżer kontekstu.

(Jak już wiesz) dunder init odpowiada za inicjalizację

Oprócz metod `__enter__` i `__exit__` do swojej klasy możesz według potrzeb dodawać też inne metody, w tym zdefiniowaną samodzielnie metodę `__init__`. Jak już wiesz z lektury poprzedniego rozdziału, definiując ten dunder, możesz przeprowadzić dodatkową inicjalizację obiektu. Metoda ta jest wykonywana *przed* metodą `__enter__` (czyli *przed wykonaniem kodu odpowiedzialnego za konfigurację Twojego menedżera kontekstu*).

Definiowanie metody `__init__` nie jest bezwzględnie wymagane w przypadku menedżera kontekstu (jako że wszystkim, czego naprawdę potrzebujesz, są metody `__enter__` i `__exit__`), czasami jednak warto o to zadbać, ponieważ można dzięki temu odseparować wszelkie działania związane z inicjalizacją od działań typowo konfiguracyjnych. Gdy będziemy tworzyć menedżer kontekstu, który ma być wykorzystywany z naszym połączeniem z bazą danych (co będzie miało miejsce w dalszej części tego rozdziału), zdefiniujemy metodę `__init__`, aby zainicjalizować dane opisujące połączenie z naszą bazą. Nie jest to absolutnie konieczne, ale pomoże naszym zdaniem zachować należyty porządek i sprawi, że kod klasy menedżera kontekstu łatwiej będzie przeczytać i zrozumieć.

Menedżer kontekstu widziałeś już w akcji

Z instrukcją `with` miałeś po raz pierwszy do czynienia w rozdziale 6., gdy korzystałeś z niej, aby zapewnić, żeby otwarty wcześniej plik był *automatycznie* zamykany po tym, gdy zakończy się związaną z nim instrukcja `with`. Przypomnij sobie, jak za pomocą przedstawionego poniżej kodu otwierałeś plik *todos.txt*, a następnie odczytywałeś z niego poszczególne wiersze danych i wyświetlałeś je jeden po drugim na ekranie, zanim plik został automatycznie zamknięty (dzięki temu, że wcześniej był otwarty za pomocą menedżera kontekstu).

To jest Twoja pierwsza w historii instrukcja with (pożyczona z rozdziału 6.).

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

Przyjrzyjmy się raz jeszcze tej instrukcji `with`, wskazując miejsca, w których wywoływane są dundery `enter`, `exit` oraz `init`. Ponumerowaliśmy poszczególne komentarze, aby ułatwić Ci zrozumienie kolejności, w jakiej te metody są wykonywane. Zauważ, że nigdzie nie widać tu kodu odpowiedzialnego za konfigurację, wykonanie i demontaż; wiemy jedynie (i szczerze w to wierzymy), że odpowiednie metody są wywoływane „w tle", gdy jest to konieczne.

1. Gdy interpreter natrafia na tę instrukcję with, zaczyna od wywołania dundera init związanego z tym wywołaniem funkcji open.

2. Gdy tylko dunder init zostanie wykonany, interpreter wywołuje dunder enter, aby zapewnić, że wynik wywołania funkcji open zostanie przypisany do zmiennej tasks.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

3. Gdy instrukcja with kończy swoje działanie, interpreter wywołuje należący do menedżera kontekstu dunder exit, aby posprzątać. W tym przykładzie interpreter zapewnia, że otwarty plik zostanie poprawnie zamknięty, zanim stanie się cokolwiek innego.

Czego wymaga się od Ciebie

Zanim zaczniemy tworzyć własny menedżer kontekstu (z pomocą nowej klasy), przejrzyjmy, czego oczekuje od Ciebie protokół zarządzania kontekstem, abyś mógł skorzystać z możliwości instrukcji `with`. Musisz utworzyć klasę, która zapewnia:

1. metodę `__init__`, aby przeprowadzić inicjalizację (jeśli jest to konieczne);
2. metodę `__enter__`, aby dokonać wszelkiej konfiguracji;
3. metodę `__exit__`, aby wykonać wszelki demontaż (czyli sprzątanie).

Uzbrojeni w tę wiedzę, utwórzmy teraz klasę menedżera kontekstu, pisząc te metody jedna po drugiej i zapożyczając nasz istniejący kod obsługi bazy danych, gdy tylko będzie to konieczne.

Utwórz nową klasę menedżera kontekstu

Aby ruszyć dalej, musimy nadać naszej klasie nazwę. Poza tym umieśćmy jej kod w osobnym pliku, aby dało się go łatwo ponownie wykorzystać (pamiętaj, że gdy umieszczasz kod Pythona w oddzielnym pliku, staje się on modułem, który można importować w innych programach wedle potrzeb).

Nasz nowy plik nazwijmy *DBcm.py* (to skrót od słów *database context manager*, czyli „menedżer kontekstu bazy danych"), a nowej klasie nadajmy nazwę `UseDatabase`. Pamiętaj, aby utworzyć ten plik w tym samym folderze, w którym znajduje się obecnie nasza aplikacja WWW, ponieważ w jej kodzie trzeba będzie zaimportować klasę `UseDatabase` (oczywiście gdy już napiszesz jej kod).

Pamiętaj, aby nazywając klasę Pythona, korzystać z zapisu CamelCase.

Posługując się swoim ulubionym edytorem tekstowym (lub środowiskiem IDLE), otwórz nowe okno edycji i zapisz nowy, pusty plik pod nazwą *DBcm.py*. Wiesz już, że aby Twoja klasa spełniała wymogi protokołu zarządzania kontekstem, musi zapewniać:

1. metodę `__init__`, która odpowiada za inicjalizację;
2. metodę `__enter__`, która zawiera wszelki kod konfiguracyjny;
3. metodę `__exit__`, która zawiera wszelki kod demontażowy.

Teraz dodajmy do kodu klasy „puste" definicje każdej z tych trzech wymaganych metod. Pusta metoda zawiera pojedynczą instrukcję `pass`. Poniżej znajdziesz kod, który udało nam się opracować

Oto jak wygląda plik DBcm.py otwarty w oknie edytora IDLE. W tej chwili składa się z jednej instrukcji importu oraz definicji klasy UseDatabase, która zawiera trzy „puste" metody.

DBcm.py - C:/Users/lsuma/Desktop/Python/web...

File Edit Format Run Options Window Help

```
import mysql.connector


class UseDatabase:

    def __init__(self):
        pass

    def __enter__(self):
        pass

    def __exit__(self):
        pass
```

Ln: 14 Col: 0

Zwróć uwagę, że na początku pliku *DBcm.py* umieściliśmy instrukcję `import` włączającą możliwości rozwiązania *MySQL Connector* (od którego zależna jest nasza klasa).

Teraz musimy już tylko przenieść odpowiednie fragmenty kodu z funkcji `log_request` do właściwych metod należących do klasy `UseDatabase`. Cóż... Gdy piszemy „my", tak naprawdę mamy na myśli **Ciebie**. Czas zakasać rękawy i napisać trochę kodu metod.

Zainicjalizuj klasę przy użyciu konfiguracji bazy danych

Przypomnijmy sobie, jak mamy zamiar używać menedżera kontekstu `UseDatabase`. Oto kod z rozdziału 7. napisany w taki sposób, aby wykorzystywał instrukcję `with`, korzystającą z kolei z menedżera kontekstu `UseDatabase`, który niebawem opracujesz.

```
from DBcm import UseDatabase

dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

with UseDatabase(dbconfig) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Zaimportuj menedżer kontekstu z pliku DBcm.py.

Tutaj znajdują się właściwości połączenia z bazą danych.

Menedżer kontekstu zwraca zmienną cursor.

Menedżer kontekstu UseDatabase spodziewa się otrzymać słownik zawierający właściwości połączenia z bazą danych.

Ten kod ma taką samą postać jak wcześniej.

Zaostrz ołówek

Zacznijmy od metody `__init__`, z której będziemy korzystać w celu inicjalizacji wszystkich atrybutów klasy `UseDatabase`. Na podstawie przedstawionego powyżej sposobu użycia klasy możemy stwierdzić, że metoda `__init__` przyjmuje jeden argument, którym jest słownik właściwości połączenia o nazwie `config` (a który będziesz musiał dodać do pokazanego poniżej wiersza `def`). Sprawmy, aby słownik ten był zapisywany jako atrybut o nazwie `configuration`. Dodaj kod niezbędny do zapisania słownika w tym atrybucie do metody `__init__`.

```
import mysql.connector

class UseDatabase:

    def __init__(self, ..................................... )

    ..............................................................
```

Uzupełnij wiersz def.

Czy czegoś tu brakuje?

Zapisz słownik konfiguracji w atrybucie.

Zaostrz ołówek
Rozwiązanie

Zaczęliśmy od metody `__init__`, której zadaniem jest inicjalizacja wszystkich atrybutów klasy `UseDatabase`. Metoda ta przyjmuje jeden argument, którym jest słownik właściwości połączenia o nazwie `config` (a który musiałeś dodać do pokazanego poniżej wiersza `def`). Miałeś sprawić, aby słownik ten był zapisywany jako atrybut o nazwie `configuration`. Miałeś dodać kod niezbędny do zapisania słownika w tym atrybucie do metody `__init__`.

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict ) -> None :
        self.configuration = config
```

Dunder init przyjmuje jeden argument będący słownikiem, któremu nadaliśmy nazwę „config".

Wartość argumentu config zostaje przypisana do atrybutu o nazwie „configuration". Czy pamiętałeś o tym, aby poprzedzić go słowem self?

(Opcjonalna) adnotacja None potwierdza, że ta metoda nie zwraca żadnej wartości (o czym dobrze wiedzieć). Wiersz def kończy się dwukropkiem.

Twój menedżer kontekstu nabiera kształtu

Po opracowaniu kodu metody `__init__` możesz przystąpić do pisania kodu metody `__enter__`. Zanim to zrobisz, sprawdź, czy — jak dotąd — Twój kod odpowiada naszemu, który został poniżej przedstawiony w oknie edycji środowiska IDLE.

Upewnij się, że Twój kod metody __init__ jest taki jak nasz.

DBcm.py - C:/Users/lsuma/Desktop/Python/web...

File Edit Format Run Options Window Help

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self):
        pass

    def __exit__(self):
        pass
```

Ln: 14 Col: 0

Dokonaj konfiguracji za pomocą metody __enter__

Dunder `enter` stanowi miejsce, w którym powinieneś wstawić kod odpowiedzialny za wszelkie działania związane z konfiguracją, jakie należy wykonać *przed* uruchomieniem zestawu instrukcji `with`. Przypomnij sobie spełniający to zadanie fragment kodu funkcji `log_request`.

Tutaj znajduje się kod konfiguracyjny wchodzący w skład funkcji log_request.

```
...
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

conn = mysql.connector.connect(**dbconfig)
cursor = conn.cursor()

 _SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
...
```

Ten kod konfiguracyjny wykorzystuje słownik właściwości połączenia, aby połączyć się z bazą danych MySQL, a następnie tworzy kursor bazy danych dla tego połączenia (który jest nam potrzebny do wysyłania poleceń do bazy z poziomu kodu Pythona). Jako że kod ten będziesz pisał za każdym razem, gdy będziesz implementował komunikację ze swoją bazą danych, włączmy go do klasy menedżera kontekstu, dzięki czemu będzie go można łatwiej użyć ponownie.

Zaostrz ołówek

Metoda `__enter__` musi korzystać z właściwości konfiguracji przechowywanych w atrybucie `self.configuration`, aby można było nawiązać połączenie z bazą danych i utworzyć kursor. Poza obowiązkowym argumentem `self` metoda ta nie potrzebuje żadnych innych argumentów, powinna jednak zwracać kursor. Uzupełnij przedstawiony poniżej kod metody `__enter__`.

Czy masz pomysł na odpowiednią adnotację?

Tutaj dodaj kod konfiguracyjny.

```
def __enter__(self) ..................................:
    ..............................................................
    ..............................................................
    return ..............................................................
```

Nie zapomnij o zwróceniu kursora.

Zaostrz ołówek
Rozwiązanie

Metoda `__enter__` korzysta z właściwości konfiguracji przechowywanych w atrybucie `self.configuration`, aby można było nawiązać połączenie z bazą danych i utworzyć kursor. Poza obowiązkowym argumentem `self` metoda ta nie potrzebuje żadnych innych argumentów, powinna jednak zwracać kursor. Miałeś uzupełnić przedstawiony poniżej kod metody `__enter__`.

```
def __enter__(self)      -> 'cursor'      :
    self.conn = mysql.connector.connect(**self.configuration)
    self.cursor = self.conn.cursor()
    return  self.cursor
```

Czy pamiętałeś, aby poprzedzić wszystkie atrybuty słowem self?

Ta adnotacja informuje użytkowników Twojej klasy, czego mogą się spodziewać jako wartości zwracanej przez tę metodę.

Pamiętaj, aby odwołać się tu do atrybutu self.configuration zamiast do zmiennej dbconfig.

Zwróć kursor.

Nie zapomnij o poprzedzeniu nazw wszystkich atrybutów słowem self

Być może zaskoczyło Cię, że tworząc naszą metodę `__enter__`, zmienne `conn` i `cursor` postanowiliśmy uczynić atrybutami klasy (poprzez poprzedzenie każdego z nich słowem `self`). Zrobiliśmy to, aby zapewnić, że wartości te przetrwają zakończenie działania metody `__enter__`, jako że obydwie są potrzebne w metodzie `__exit__`. Aby mieć pewność, że tak właśnie będzie, dodaliśmy przedrostek `self` do nazw zmiennych `conn` i `cursor`, dopisując je w ten sposób do listy atrybutów klasy.

Zanim zabierzesz się do pisania metody `__exit__`, przekonaj się, czy Twój kod nadal odpowiada przedstawionemu poniżej naszemu kodowi.

Już niemal skończyłeś. Została Ci tylko jedna metoda do napisania.

DBcm.py - C:/Users/lsuma/Desktop/Python/webapp/DBcm.py (3.6.0)

File Edit Format Run Options Window Help

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self):
        pass
```

Ln: 16 Col: 0

Przeprowadź demontaż za pomocą metody __exit__

Dunder `exit` to miejsce na kod demontujący, który powinien być wykonany, gdy Twoja instrukcja `with` kończy swoje działanie. Przypomnij sobie fragment kodu funkcji `log_request`, który odpowiadał za wymagane tu działania. Został on przedstawiony poniżej.

```
    ...
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))
conn.commit()
cursor.close()
conn.close()
```

To jest kod demontujący.

Kod demontujący powoduje zatwierdzenie wszystkich danych do zapisu w bazie, a następnie zamyka kursor oraz połączenie. Demontaż ten odbywa się za *każdym* razem, gdy komunikujesz się ze swoją bazą danych, więc dodajmy ten fragment do klasy menedżera kontekstu, przenosząc odpowiednie trzy wiersze kodu do metody `__exit__`.

Zanim to jednak zrobisz, powinieneś się dowiedzieć, że z metodą `__exit__` wiąże się pewna komplikacja. Chodzi o to, że metoda ta musi obsłużyć wszystkie wyjątki, które mogą się pojawić w trakcie wykonywania zestawu instrukcji `with`. Gdy coś pójdzie źle, interpreter *zawsze* powiadamia o tym metodę `__exit__`, przekazując jej trzy argumenty: `exec_type`, `exc_value` oraz `exc_trace`. Tworząc wiersz `def` tej metody, musisz wziąć to pod uwagę. Z tego też powodu dodaliśmy tam trzy argumenty, o czym możesz się przekonać poniżej. Mimo to na razie mamy zamiar zupełnie *zignorować* ten mechanizm obsługi wyjątków, jednak powrócimy do niego w dalszej części niniejszego rozdziału, gdy będziemy omawiać to, co może pójść nie tak, i jak możemy sobie z tym poradzić (dlatego bądź czujny).

Zaostrz ołówek

Kod demontujący to ten, w którym powinniśmy zadbać o posprzątanie po swoich operacjach. W przypadku naszego menedżera kontekstu sprzątanie oznacza zapewnienie, że wszystkie dane są zatwierdzone w bazie danych przed zamknięciem kursora oraz połączenia. Do przedstawionej poniżej metody dodaj kod, który wydaje Ci się potrzebny.

Na razie nie przejmuj się tymi argumentami.

W tym miejscu dodaj kod demontujący.

```
def __exit__(self, exc_type, exc_value, exc_trace) ..............:
    ..............................................................
    ..............................................................
    ..............................................................
```

Zaostrz ołówek
Rozwiązanie

Kod demontujący to ten, w którym powinniśmy zadbać o posprzątanie po swoich operacjach. W przypadku naszego menedżera kontekstu sprzątanie oznacza zapewnienie, że wszystkie dane są zatwierdzone w bazie danych przed zamknięciem kursora oraz połączenia. Do przedstawionej poniżej metody miałeś dodać kod, który wydaje Ci się potrzebny.

Na razie nie przejmuj się tymi argumentami.

```
def __exit__(self, exc_type, exc_value, exc_trace) -> None:
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

-> None

Zapisane wcześniej atrybuty są wykorzystywane do zatwierdzenia niezapisanych danych, jak również zamknięcia kursora oraz połączenia. Jak zawsze pamiętaj tutaj, aby nazwy atrybutów poprzedzić przedrostkiem self.

Ta adnotacja informuje, że metoda nie zwraca wartości; adnotacje tego rodzaju są opcjonalne, ale ich dodawanie stanowi dobrą praktykę.

Twój menedżer kontekstu jest gotów do testów

Gdy napisałeś już kod metody `__exit__`, nadszedł czas na przetestowanie Twojego menedżera kontekstu, zanim będziemy mogli zintegrować go z kodem aplikacji WWW. Jak już mamy w zwyczaju, najpierw sprawdźmy ten nowy kod w wierszu poleceń powłoki Pythona. Przedtem jednak po raz ostatni upewnij się, że Twój kod jest taki sam jak nasz.

Ukończona klasa menedżera kontekstu UseDatabase

DBcm.py - C:/Users/Isuma/Desktop/Python/webapp/DBcm.py (3.6.0)

File Edit Format Run Options Window Help

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Ln: 18 Col: 0

„Prawdziwy" kod klasy zawierałby dokumentację, ale usunęliśmy ją stąd, aby zaoszczędzić trochę miejsca (na tej stronie). Pliki z kodami przedstawionymi w tej książce, które możesz pobrać, zawsze opatrzone są komentarzami.

Jazda próbna

Przejdź do okna edycji środowiska IDLE, w którym otwarty jest kod zapisany w pliku *DBcm.py*, a następnie naciśnij klawisz *F5*, aby przetestować swój menedżer kontekstu.

Zaimportuj klasę menedżera kontekstu z pliku modułu DBcm.py.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> from DBcm import UseDatabase
>>>
>>> dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }
>>>
>>> with UseDatabase(dbconfig) as cursor:
        _SQL = """show tables"""
        cursor.execute(_SQL)
        data = cursor.fetchall()


>>> data
[('log',)]
>>>
>>>
```

Ln: 125 Col: 4

Umieść właściwości połączenia w słowniku.

Użyj menedżera kontekstu w celu wysłania zapytania SQL do serwera i odebrania wynikowych danych.

Zwrócone dane mogą wydawać Ci się trochę dziwne, dopóki nie przypomnisz sobie, że wywołanie metody cursor.fetchall zwraca listę krotek, z których każda odpowiada wierszowi wyników (zwróconych z bazy danych).

Nie ma tu zbyt wiele kodu, prawda?

Mamy nadzieję, że przyglądając się przedstawionemu powyżej kodowi, stwierdzisz, że nie wygląda on zbyt strasznie. Dzięki temu, że udało Ci się przenieść część swojego kodu obsługi bazy danych do klasy `UseDatabase`, operacje inicjalizacji, konfiguracji i demontażu odbywają się teraz „w tle" i odpowiada za nie opracowany przez Ciebie menedżer kontekstu. Musisz tu jedynie podać właściwości połączenia oraz zapytanie SQL, które chcesz wykonać, a Twój menedżer załatwi za Ciebie całą resztę. Kod odpowiedzialny za przygotowanie operacji i posprzątanie po niej jest ponownie wykorzystywany jako część menedżera kontekstu. Wyraźniej widać też teraz, co jest „treścią" tego kodu: pobranie danych z bazy i ich przetworzenie. Menedżer kontekstu ukrywa szczegóły łączenia się z bazą danych oraz rozłączania z nią (które zawsze pozostaną takie same), pozwalając Ci się skupić na tym, co naprawdę chcesz zrobić ze swoimi danymi.

Zaktualizujmy kod Twojej aplikacji WWW w taki sposób, aby wykorzystywała ten menedżer kontekstu.

Zrewiduj kod swojej aplikacji WWW (1. z 2)

Już dość dawno nie przyglądałeś się kodowi swojej aplikacji WWW.

Ostatnim razem, gdy miałeś z nim do czynienia (w rozdziale 7.) zmodyfikowałeś funkcję `log_request` w taki sposób, aby zapisywała żądania sieciowe do bazy danych MySQL. Powodem, dla którego przeszliśmy potem do prezentacji podstawowych możliwości związanych z klasami (w rozdziale 8.), była chęć ustalenia najlepszego sposobu współdzielenia kodu obsługi bazy danych, który został dodany do tej funkcji. Wiemy już teraz, że sposobem tym jest (w naszym przypadku) skorzystanie z opracowanej właśnie klasy menedżera kontekstu o nazwie `UseDatabase`.

Oprócz tego, że należy zmienić funkcję `log_request` tak, aby korzystała z menedżera kontekstu, mamy do zrobienia coś jeszcze. Inną funkcją, która powinna zacząć używać danych pochodzących z bazy, jest funkcja o nazwie `view_the_log` (korzystająca obecnie z danych zapisanych w pliku tekstowym *vsearch.log*). Zanim przystąpimy do tych działań, przypomnijmy sobie, jak w tej chwili wygląda kod aplikacji (prezentujemy go na tej i na następnej stronie). Zostały w nim wyróżnione fragmenty, które wymagają zmiany.

Kod Twojej aplikacji WWW znajduje się w pliku vsearch4web.py należącym do folderu webapp.

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

import mysql.connector

app = Flask(__name__)

def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""
    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Zamiast tego powinniśmy tutaj zaimportować moduł DBcm.

Ten kod trzeba zmodyfikować w taki sposób, aby wykorzystywał menedżer kontekstu UseDatabase.

Zrewiduj kod swojej aplikacji WWW (2. z 2)

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   """Wydobywa przesłane dane; przeprowadza wyszukiwanie; zwraca wyniki."""
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   log_request(request, results)
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   """Wyświetla formularz HTML tej aplikacji WWW."""
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

@app.route('/viewlog')
def view_the_log() -> 'html':
   """Wyświetla zawartość pliku logu w postaci tabeli HTML."""
   contents = []
   with open('vsearch.log') as log:
      for line in log:
         contents.append([])
         for item in line.split('|'):
            contents[-1].append(escape(item))
   titles = ('Dane z formularza', 'Adres klienta', 'Agent użytkownika', 'Wyniki')
   return render_template('viewlog.html',
                          the_title='Widok logu',
                          the_row_titles=titles,
                          the_data=contents,)

if __name__ == '__main__':
   app.run(debug=True)
```

Ten kod należy zmodyfikować w taki sposób, aby korzystał z danych odczytanych z bazy za pomocą menedżera kontekstu UseDatabase.

Przypomnienie funkcji log_request

Jeśli chodzi o modyfikację funkcji `log_request` w taki sposób, aby korzystała z menedżera kontekstu `UseDatabase`, to spora część pracy została już w zasadzie wykonana (jak możesz się przekonać, patrząc na przedstawiony wcześniej kod).

Przyjrzyj się pokazanej poniżej funkcji `log_request` raz jeszcze. W tej chwili słownik zawierający właściwości połączenia z bazą danych (reprezentowany w kodzie przez zmienną `dbconfig`) jest definiowany wewnątrz tej funkcji. Jako że będziesz chciał z niego korzystać również w innej funkcji, którą zamierzasz zmienić (czyli `view_the_log`), powinieneś przenieść go poza funkcję `log_request`, dzięki czemu w razie potrzeby będzie mógł być współdzielony przez inne funkcje.

Przenieśmy ten słownik poza funkcję, aby w razie potrzeby można z niego było korzystać również w innych funkcjach.

```
def log_request(req: 'flask_request', res: str) -> None:

    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Zamiast przenosić słownik `dbconfig` do globalnej przestrzeni naszej aplikacji WWW, lepiej byłoby w jakiś sposób dodać go do wewnętrznej konfiguracji tej aplikacji.

Tak się dobrze składa, że Flask (podobnie jak wiele innych frameworków sieciowych) zapewnia wbudowany mechanizm konfiguracyjny, którym jest słownik (nazywany we Flasku *app.config*) umożliwiający Ci dostosowanie pewnych wewnętrznych ustawień aplikacji WWW. Jako że *app.config* stanowi zwykły słownik języka Python, możesz do niego wedle potrzeby dodawać własne klucze i wartości. Skorzystasz z tej możliwości w przypadku danych przechowywanych w słowniku `dbconfig`.

Pozostałą część kodu funkcji `log_request` będziemy następnie mogli zmienić w taki sposób, aby korzystała z menedżera kontekstu `UseDatabase`.

Dokonajmy tych zmian teraz.

Modyfikacja funkcji log_request

Po dokonaniu odpowiednich zmian w naszej aplikacji WWW jej kod wygląda tak, jak zostało to pokazane poniżej.

Poprzednią instrukcję import zastąpiliśmy instrukcją pokazaną tutaj.

vsearch4web.py - C:\Users\lsuma\Desktop\Python\webapp\vsearch4web.py (3.6.0)

File Edit Format Run Options Window Help

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

from DBcm import UseDatabase

app = Flask(__name__)

app.config['dbconfig'] = { 'host': '127.0.0.1',
                           'user': 'vsearch',
                           'password': 'vsearchpasswd',
                           'database': 'vsearchlogDB', }


def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""

    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

Ln: 29 Col: 0

Do konfiguracji aplikacji WWW dodaliśmy słownik zawierający właściwości połączenia.

Zmodyfikowaliśmy ten kod, aby wykorzystywał klasę UseDatabase. Użyliśmy przy tym konfiguracji połączenia z bazą danych zapisanej w słowniku app.config.

Znajdującą się w okolicach początku pliku wcześniejszą instrukcję `import mysql.connector` zastąpiliśmy nową instrukcją importowania, która wyciąga z modułu `DBcm` klasę `UseDatabase`. Mogliśmy sobie na to swobodnie pozwolić, ponieważ sam plik *DBcm.py* zawiera już w swoim kodzie instrukcję `import mysql.connector` (a nie chcemy przecież importować tego modułu dwukrotnie).

Przenieśliśmy również do konfiguracji aplikacji WWW słownik zawierający właściwości połączenia z bazą danych. Ponadto zmodyfikowaliśmy kod funkcji `log_request` w taki sposób, aby korzystała ona z menedżera kontekstu.

Dzięki swoim dotychczasowym doświadczeniom z używaniem klas i menedżerów kontekstu powinieneś już być w stanie przeczytać i zrozumieć przedstawiony powyżej kod.

Przystąpmy teraz do modyfikacji funkcji `view_the_log`. Przed przejściem do następnej strony upewnij się, że Twój kod aplikacji WWW wygląda dokładnie tak samo jak nasz.

Przypomnienie funkcji view_the_log

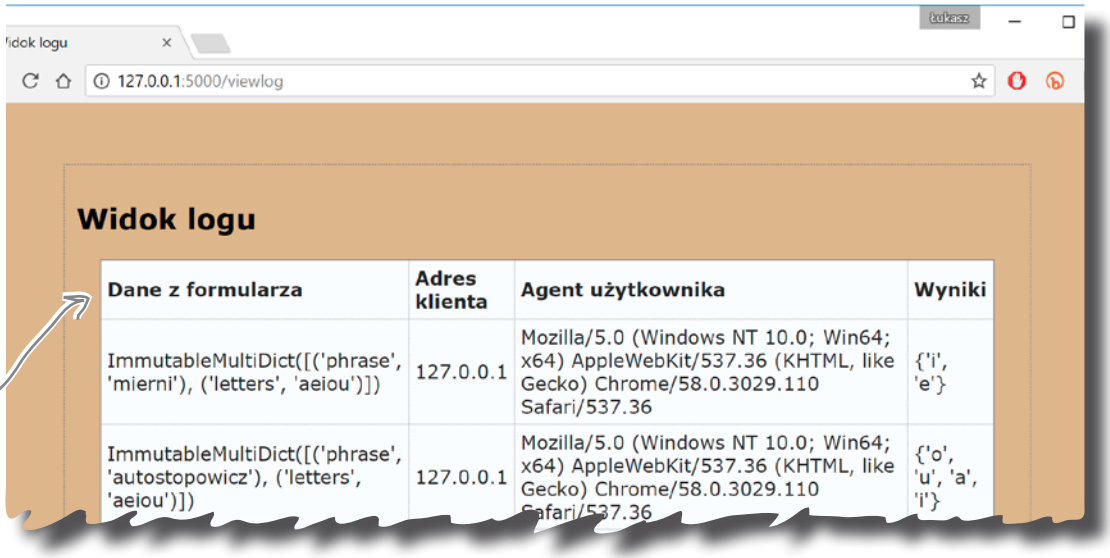
Przyjrzyjmy się teraz naprawdę dokładnie kodowi funkcji `view_the_log`, jako że nie oglądaliśmy go uważnie już dość długo. Tytułem przypomnienia: bieżąca wersja tej funkcji wydobywa zalogowane dane z pliku tekstowego noszącego nazwę *vsearch.log*, przekształca je do postaci listy list (o nazwie `contents`), a następnie przekazuje do szablonu nazwanego *viewlog.html*, tak jak zostało to przedstawione poniżej.

Weź każdy wiersz danych z pliku, a następnie przekształć go w listę odpowiednio sformatowanych elementów, które są dodawane do listy contents.

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))

    titles = ('Dane z formularza', 'Adres klienta', 'Agent użytkownika', 'Wyniki')
    return render_template('viewlog.html',
                           the_title='Widok logu',
                           the_row_titles=titles,
                           the_data=contents,)
```

Przetwarzane dane logu są przesyłane do szablonu w celu wyświetlenia.

Poniżej możesz zobaczyć efekt powstały, gdy szablon *viewlog.html* zostaje wyrenderowany z danymi pochodzącymi z listy list `contents`. Twoja aplikacja WWW udostępnia tę stronę pod URL-em `/viewlog`.



Dane z formularza	Adres klienta	Agent użytkownika	Wyniki
ImmutableMultiDict([('phrase', 'mierni'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'i', 'e'}
ImmutableMultiDict([('phrase', 'autostopowicz'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36	{'o', 'u', 'a', 'i'}

Dane zapisane na liście contents są wyświetlane na tej stronie. Zauważ, że dane pochodzące z formularza (wartości zmiennych phrase oraz letters) są prezentowane w jednej kolumnie.

Nie tylko kod się tu zmienia

Zanim weźmiemy się za zmianę kodu funkcji `view_the_log` w taki sposób, aby korzystała z opracowanego wcześniej menedżera kontekstu, zatrzymajmy się na chwilę, aby przyjrzeć się danym przechowywanym w tabeli `log` Twojej bazy danych. Gdy w rozdziale 7. testowałeś pierwotną wersję funkcji `log_request`, mogłeś zalogować się do konsoli MySQL, a następnie sprawdzić, że dane zostały zapisane w bazie. Przypomnij sobie tę sesję (została pokazana poniżej).

```
Administrator: Wiersz polecenia

C:\>mysql -u vsearch -p vsearchlogDB
Enter password: *************
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 24
Server version: 5.7.18-log MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select * from log;
+----+---------------------+---------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase        | letters | ip        | browser_string | results              |
+----+---------------------+---------------+---------+-----------+----------------+----------------------+
|  1 | 2017-06-20 12:40:06 | autosto ... cz | aeiou   | 127.0.0.1 | firefox        | {'u', 'a', 'o', 'i'} |
|  2 | 2017-06-20 12:40:51 | mierni        | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2017-06-21 12:58:52 | galaktyka     | tym     | 127.0.0.1 | chrome         | {'y', 't'}           |
|  4 | 2017-06-21 13:02:14 | mierni        | vxy     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+---------------+---------+-----------+----------------+----------------------+
4 rows in set (0.00 sec)

mysql> quit
Bye
```

Dane logu zapisane w tabeli bazy danych

Gdy przyjrzysz się przedstawionym powyżej danym i odniesiesz je do tego, co znajduje się obecnie w pliku *vsearch.log*, łatwo dojdziesz do wniosku, że część przetwarzania wykonywanego przez funkcję `view_the_log` nie jest już potrzebna, ponieważ odpowiednie dane są teraz przechowywane w tabeli. Poniżej możesz zobaczyć przykładowy wycinek danych zapisanych w pliku *vsearch.log*.

```
ImmutableMultiDict([('phrase', 'galaktyka'), ('letters', 'tym')])|127.0.0.1|Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36|{'t', 'y'}
```

Dane logu zapisane w postaci jednego łańcucha znakowego w pliku vsearch.log

Część kodu należącego obecnie do funkcji `view_the_log` znajduje się w niej wyłącznie dlatego, że dane logu przechowywane są w tej chwili w postaci kolekcji długich łańcuchów znakowych (rozdzielonych znakami pionowej kreski) w pliku *vsearch.log*. Format ten sprawdził się całkiem nieźle, ale musieliśmy napisać trochę dodatkowego kodu, aby zapewnić sobie możliwość obsłużenia danych.

Nie mamy tego problemu w przypadku danych zapisanych w tabeli `log`, ponieważ z definicji mają one pewną strukturę. Powinno to oznaczać, że w funkcji `view_the_log` nie musimy wykonywać żadnych dodatkowych działań, lecz jedynie wydobyć dane z tabeli, które — na szczęście — zwracane są w postaci listy krotek (co zawdzięczamy metodzie `fetchall` zapewnianej przez DB-API).

Poza tym w tabeli `log` wartości zmiennych `phrase` i `letters` są rozdzielone. Gdy wprowadzisz niewielką zmianę w kodzie renderującym szablon, łatwo możesz uzyskać efekt w postaci pięciu oddzielnych kolumn danych (zamiast czterech, jak ma to miejsce w tej chwili), dzięki czemu wynikowa strona wyświetlana przez przeglądarkę internetową będzie pokazywać bardziej użyteczne i czytelne informacje.

Modyfikacja funkcji view_the_log

Na podstawie informacji, które przedstawiliśmy Ci na kilku ostatnich stronach, można wskazać dwie rzeczy wymagające zmiany w bieżącym kodzie funkcji `view_the_log`. Oto niezbędne zmiany:

1. Pobierz dane logu z tabeli bazy danych (zamiast z pliku).
2. Zmień listę `titles` w taki sposób, aby odpowiadała pięciu kolumnom danych (zamiast czterem).

Jeśli drapiesz się teraz w głowę i zastanawiasz się, dlaczego ta krótka lista zmian nie obejmuje dostosowania szablonu *viewlog.html*, nie łam sobie już nad tym głowy, ponieważ nie musisz dokonywać żadnych zmian w *tym* pliku. Dzieje się tak dlatego, że szczęśliwie dla nas aktualny szablon przetwarza taką liczbę tytułów oraz taką ilość danych, jaką mu przekażesz.

Poniżej przedstawiony został bieżący kod funkcji `view_the_log`, który zaraz zaczniesz modyfikować.

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))

    titles = ('Dane z formularza', 'Adres klienta', 'Agent użytkownika', 'Wyniki')
    return render_template('viewlog.html',
                           the_title='Widok logu',
                           the_row_titles=titles,
                           the_data=contents,)
```

W wyniku wykonania przedstawionego powyżej zadania nr 1 ten fragment kodu zostanie zastąpiony innym.

W wyniku wykonania przedstawionego powyżej zadania nr 2 ten wiersz ulegnie zmianie.

Oto zapytanie SQL, którego będziesz potrzebował

Przed przystąpieniem do wykonania kolejnego ćwiczenia (w którym zaktualizujesz funkcję `view_the_log`) przedstawimy Ci zapytanie SQL, którego wykonanie spowoduje zwrócenie wszystkich zalogowanych danych przechowywanych w bazie danych MySQL związanej z aplikacją WWW. Dane te zostaną zwrócone do kodu Pythona w postaci listy krotek. W ćwiczeniu zaprezentowanym na następnej stronie powinieneś skorzystać z zapytania SQL przedstawionego poniżej.

```
select phrase, letters, ip, browser_string, results
from log
```

Zaostrz ołówek

Poniżej znajdziesz kod funkcji `view_the_log`, która powinna zostać zmieniona w taki sposób, aby korzystała z danych zapisanych w tabeli `log`. Twoim zadaniem jest dopisanie brakującego kodu. Pamiętaj o przeczytaniu zamieszczonych poniżej komentarzy, które wskażą Ci, co masz zrobić.

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    with ..................................................................... :

        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""

        ..........................................................................
        ..........................................................................

    titles = ( ..................., ....................., 'Adres klienta', 'Agent użytkownika',
'Wyniki')
    return render_template('viewlog.html',
                           the_title='Widok logu',
                           the_row_titles=titles,
                           the_data=contents,)
```

Tutaj użyj swojego menedżera kontekstu i nie zapomnij o kursorze.

Prześlij zapytanie do serwera, a następnie odbierz wyniki.

Których nazw kolumn tu brakuje?

Zrobię sobie tylko notatkę na temat tego, co się tu dzieje. Mój nowy kod jest nie tylko krótszy niż to, co miałam wcześniej, lecz teraz łatwiej go także przeczytać i zrozumieć.

Tak, właśnie to przez cały czas było naszym celem.

Przenosząc dane logu do bazy MySQL, pozbyłeś się konieczności tworzenia, a następnie przetwarzania plików korzystających z własnego formatu tekstowego.

Co więcej, ponownie używając menedżera kontekstu, uprościłeś swoją komunikację z MySQL-em w Pythonie. Cóż mogłoby się tu nie podobać?

Zaostrz ołówek
Rozwiązanie

Poniżej przedstawiliśmy kod funkcji `view_the_log`, która miała zostać zmieniona w taki sposób, aby korzystała z danych zapisanych w tabeli `log`. Twoim zadaniem było dopisanie brakującego kodu.

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    with UseDatabase(app.config['dbconfig']) as cursor:

        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()

    titles = ('Fraza', 'Litery', 'Adres klienta', 'Agent użytkownika', 'Wyniki')

    return render_template('viewlog.html',
                           the_title='Widok logu',
                           the_row_titles=titles,
                           the_data=contents,)
```

Ten wiersz kodu jest taki sam jak w funkcji log_request.

Prześlij zapytanie do serwera, a następnie odbierz wyniki. Zwróć uwagę, że otrzymane dane zostały przypisane do zmiennej contents.

Dodaj właściwe nazwy kolumn.

To już niemal czas na ostatnią jazdę próbną

Zanim weźmiemy w obroty tę nową wersję Twojej aplikacji WWW, poświęć chwilę, aby się upewnić, że Twoja funkcja `view_the_log` jest identyczna z przedstawioną poniżej naszą funkcją.

vsearch4web.py - C:\Users\lsuma\Desktop\Python\webapp\vsearch4web.py (3.6.0)

File Edit Format Run Options Window Help

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    """Wyświetla zawartość pliku logu w postaci tabeli HTML."""
    with UseDatabase(app.config['dbconfig']) as cursor :
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
    return render_template('viewlog.html',
                           the_title='Widok logu',
                           the_row_titles=titles,
                           the_data=contents,)
```

Ln: 63 Col: 26

Jazda próbna

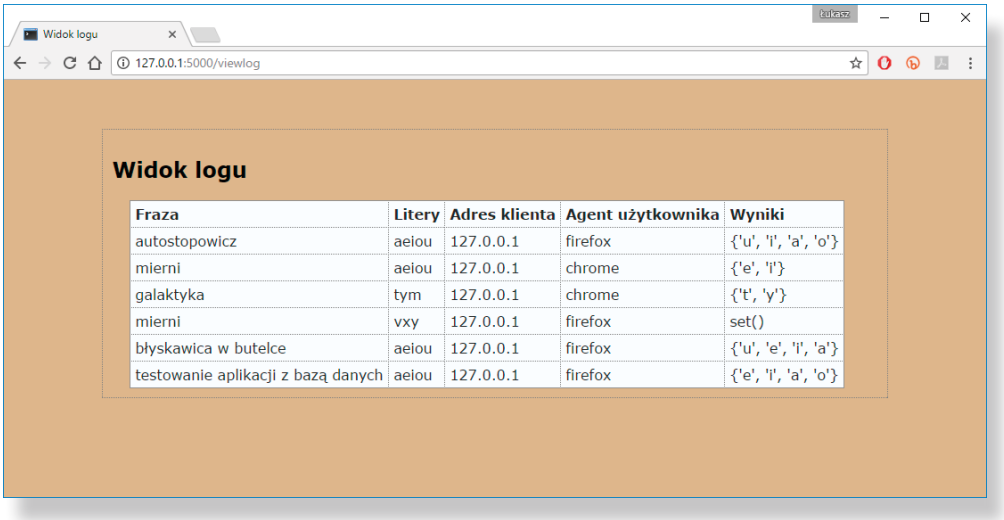
Nadszedł wreszcie czas na wzięcie w obroty Twojej aplikacji WWW gotowej do współpracy z bazą danych.

Upewnij się, że plik *DBcm.py* znajduje się w tym samym folderze, co plik *vsearch4web.py*, a następnie uruchom swoją aplikację w sposób odpowiedni dla używanego systemu operacyjnego, a więc za pomocą jednego z przedstawionych niżej poleceń:

- Użyj polecenia `python3 vsearch4web.py` w systemie *Linux* lub *Mac OS X*.
- Użyj `py -3 vsearch4web.py` w systemie *Windows*.

Skorzystaj ze swojej przeglądarki internetowej, aby przejść na stronę główną aplikacji WWW (podając w pasku adresu ciąg **`http://127.0.0.1:5000`**), a następnie przeprowadź kilka wyszukiwań. Gdy przekonasz się już, że aplikacja działa zgodnie z oczekiwaniami, skorzystaj z URL-a `/viewlog`, aby przejrzeć zawartość swojego logu w oknie przeglądarki.

Choć Twoje wyszukiwania z pewnością będą różnić się od naszych, powinieneś w tym momencie ujrzeć wyniki podobne do tych, które uzyskaliśmy my, a które przedstawione zostały poniżej. Jak widać, wszystko działa prawidłowo.



Fraza	Litery	Adres klienta	Agent użytkownika	Wyniki
autostopowicz	aeiou	127.0.0.1	firefox	{'u', 'i', 'a', 'o'}
mierni	aeiou	127.0.0.1	chrome	{'e', 'i'}
galaktyka	tym	127.0.0.1	chrome	{'t', 'y'}
mierni	vxy	127.0.0.1	firefox	set()
błyskawica w butelce	aeiou	127.0.0.1	firefox	{'u', 'e', 'i', 'a'}
testowanie aplikacji z bazą danych	aeiou	127.0.0.1	firefox	{'e', 'i', 'a', 'o'}

Uzyskany tu efekt potwierdza, że zalogowane przez nas dane zostały prawidłowo odczytane z bazy danych MySQL, gdy korzystaliśmy z URL-a `/viewlog`. Oznacza to, że kod funkcji `view_the_log` działa poprawnie, co przy okazji potwierdza również prawidłowe działanie funkcji `log_request`, jako że umieszcza ona dane logu w bazie w wyniku każdej udanej operacji wyszukiwania.

Jeśli czujesz taką potrzebę, poświęć kilka minut na zalogowanie się do swojej bazy MySQL za pomocą konsoli, aby potwierdzić, że dane są bezpiecznie przechowywane przez Twój serwer. (Albo po prostu nam zaufaj; na podstawie tego, co nasza aplikacja WWW wyświetla w oknie przeglądarki, jak widać powyżej, możemy z całą pewnością stwierdzić, że wszystko jest na swoim miejscu).

Wszystko, co pozostało...

Nadszedł czas, aby powrócić do pytań dotyczących działania naszej aplikacji WWW, które zostały po raz pierwszy postawione w rozdziale 7.:

- *Na ile żądań udało jej się odpowiedzieć?*
- *Jaka jest najczęściej używana lista liter?*
- *Z których adresów IP nadchodzą żądania?*
- *Z której przeglądarki internetowej najczęściej korzystają użytkownicy naszej aplikacji?*

Chociaż możliwe *jest* opracowanie kodu Pythona, za pomocą którego uzyskalibyśmy odpowiedzi na te pytania, nie mamy zamiaru tego robić, mimo że ten i poprzedni rozdział w całości poświęciliśmy na zapoznanie się z tym, jak język Python współpracuje z bazami danych. Naszym zdaniem pisanie tego rodzaju kodu niemal zawsze okazuje się złym posunięciem...

Jeśli zatem nie skorzystam z Pythona, aby udzielić odpowiedzi na te pytania, czego powinnam użyć? Czytając rozdział 7., nauczyłam się nieco o bazach danych i języku SQL. Czy zapytania SQL się tu sprawdzą?

SQL to zdecydowanie właściwy kierunek.

Na tego rodzaju „pytania o dane” najłatwiej jest odpowiedzieć, korzystając z mechanizmu zapytań używanej bazy danych (którym w przypadku bazy MySQL jest język SQL). Jak przekonasz się już na następnej stronie, raczej nie udałoby Ci się opracować kodu Pythona w tak krótkim czasie, jaki zajmie Ci napisanie niezbędnych zapytań SQL.

Umiejętność rozróżnienia sytuacji, w których należy używać języka Python, od tych, w których *nie* należy tego robić, jest bardzo ważna, ponieważ jest równoznaczna z wiedzą na temat tego, co odróżnia Pythona od wielu innych języków i technologii programowania. Podczas gdy większość popularnych języków zapewnia możliwość używania klas i obiektów, tylko nieliczne zapewniają mechanizm zbliżony możliwościami do protokołu zarządzania kontekstem charakterystycznego dla Pythona. (W następnym rozdziale poznasz kolejną cechę, która odróżnia go od wielu innych języków programowania; są nią dekoratory funkcji).

Zanim przejdziemy do następnego rozdziału, przyjrzymy się szybko (na jednej stronie) wspomnianym powyżej zapytaniom SQL...

Odpowiadanie na pytania o dane

Odpowiedzmy na pytania postawione w rozdziale 7. po kolei, korzystając przy tym z zapytań bazodanowych napisanych w języku SQL.

Na ile żądań udało się odpowiedzieć?

Jeśli jesteś magikiem (lub magiczką) od SQL-a, zapewne wyśmiałeś to pytanie, mając świadomość, że czegoś prostszego nie dałoby się już wymyślić. Wiesz, że przedstawione poniżej najprostsze z zapytań języka SQL umożliwia wyświetlenie wszystkich danych przechowywanych w tabeli bazy danych.

```
select * from log;
```

Aby przekształcić to zapytanie w takie, które poinformuje nas, ile wierszy danych znajduje się w tabeli, wystarczy skorzystać z funkcji SQL o nazwie `count`, której trzeba przekazać argument *.

```
select count(*) from log;
```

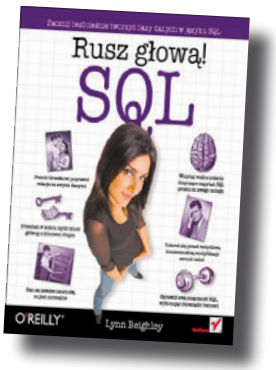
NIE prezentujemy Ci tu odpowiedzi. Jeśli chcesz je zobaczyć, musisz samodzielnie uruchomić te zapytania w konsoli MySQL (informacje na temat tego, jak to zrobić, znajdziesz w rozdziale 7.).

Jaka jest najczęściej używana lista liter?

Zapytanie SQL, które umożliwia nam uzyskanie odpowiedzi na to pytanie, wygląda dość przerażająco, jednak w rzeczywistości nie jest aż tak straszne, jak może się wydawać na pierwszy rzut oka. Możesz je znaleźć poniżej.

```
select count(letters) as 'count', letters
from log
group by letters
order by count desc
limit 1;
```

Jak już wspominaliśmy w rozdziale 7., wszystkim rozpoczynającym swoją przygodę z SQL-em (oraz tym, którzy pragną odświeżyć nieco przykurzoną, posiadaną już wiedzę) zawsze polecamy tę książkę.



Z których adresów IP nadchodzą żądania?

Wszyscy SQL-owi magicy i magiczki czytający tę książkę myślą sobie teraz z pewnością, że to zbyt proste, żeby było prawdziwe. Odpowiednie zapytanie znajdziesz poniżej.

```
select distinct ip from log;
```

Z której przeglądarki internetowej najczęściej korzystają użytkownicy naszej aplikacji?

Zapytanie SQL, które pozwala nam otrzymać odpowiedź na to pytanie, niewiele różni się od tego, które umożliwiło nam znalezienie odpowiedzi na drugie z naszych pytań. Przedstawiamy je poniżej.

```
select browser_string, count(browser_string) as 'count'
from log
group by browser_string
order by count desc
limit 1;
```

Odpowiedzi na wszystkie nurtujące Cię pytania można udzielić, korzystając z przedstawionych powyżej kilku prostych zapytań SQL. Śmiało, przejdź teraz do konsoli MySQL i zanim rozpoczniesz lekturę następnego rozdziału, wykonaj te zapytania w jej wierszu poleceń.

Kod przykładów z rozdziału 9. (1. z 2)

To jest kod menedżera kontekstu zapisany w pliku DBcm.py.

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

To jest pierwsza część kodu naszej aplikacji WWW znajdującego się w pliku vsearch4web.py.

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

from DBcm import UseDatabase

app = Flask(__name__)

app.config['dbconfig'] = { 'host': '127.0.0.1',
                           'user': 'vsearch',
                           'password': 'vsearchpasswd',
                           'database': 'vsearchlogDB', }


def log_request(req: 'flask_request', res: str) -> None:
    """Loguje szczegóły żądania sieciowego oraz wyniki."""

    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

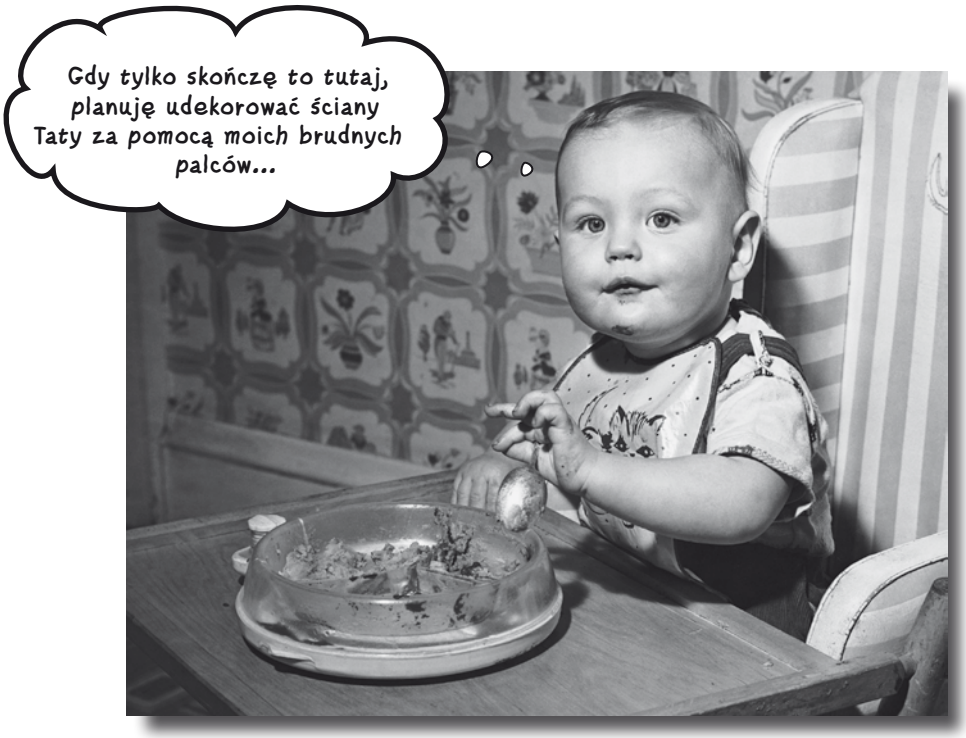
Kod przykładów z rozdziału 9. (2. z 2)

To jest druga część kodu naszej aplikacji WWW znajdującego się w pliku vsearch4web.py.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   log_request(request, results)
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

@app.route('/viewlog')
def view_the_log() -> 'html':
   with UseDatabase(app.config['dbconfig']) as cursor :
      _SQL = """select phrase, letters, ip, browser_string, results
                from log"""
      cursor.execute(_SQL)
      contents = cursor.fetchall()
   titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
   return render_template('viewlog.html',
                          the_title='Widok logu',
                          the_row_titles=titles,
                          the_data=contents,)


if __name__ == '__main__':
   app.run(debug=True)
```

10. Dekoratory funkcji

Opakowywanie funkcji



Jeśli chodzi o zwiększanie możliwości Twojego kodu, przedstawiony w rozdziale 9. protokół zarządzania kontekstem nie jest jedyną dostępną opcją.

Język Python umożliwia Ci również korzystanie z **dekoratorów** funkcji, czyli techniki, dzięki której możesz dodawać kod do istniejących funkcji *bez* konieczności modyfikacji jakiegokolwiek kodu należącego do nich. Jeśli sądzisz, że masz tu do czynienia z jakimś rodzajem czarnej magii, nie załamuj się, ponieważ to nic z tych rzeczy. Mimo to wśród różnych technik programowania tworzenie dekoratorów funkcji przez wielu programistów Pythona jest nierzadko uważane za trudne rozwiązanie i z tego też powodu nie jest wykorzystywane tak często, jak powinno być. W tym rozdziale mamy zamiar pokazać Ci, że choć stanowi to zaawansowaną technikę, tworzenie i używanie własnych dekoratorów wcale nie jest aż tak trudne, jak może się niektórym wydawać.

Twoja aplikacja WWW działa dobrze, ale...

Pokazałeś ostatnią wersję swojej aplikacji WWW kolegom, a Ci byli naprawdę pod wrażeniem tego, co zrobiłeś. Zadali Ci jednak dość frapujące pytanie: *czy mądrze jest pokazywać stronę z logami zwykłym użytkownikom sieci?*

Chodziło im o to, że każda osoba, która wie o istnieniu URL-a `/viewlog`, może z niego skorzystać, aby przeglądać zalogowane dane, niezależnie od tego, czy ma na to Twoją zgodę, czy też nie. W rzeczywistości obecnie wszystkie URL-e wykorzystywane w Twojej aplikacji mają charakter publiczny, dlatego każdy użytkownik sieci może uzyskać do nich dostęp.

W zależności od tego, co chcesz osiągnąć za pomocą swojej aplikacji WWW, może to stanowić problem lub też nie. Serwisy WWW często jednak wymagają od użytkowników autoryzacji, zanim uzyskają oni dostęp do określonych treści. Z pewnością warto zachować ostrożność w kwestii udzielania dostępu do URL-a `/viewlog`. Powstaje zatem pytanie: *jak można ograniczyć dostęp do wybranych stron Twojej aplikacji WWW?*

Dostęp mogą mieć tylko uwierzytelnieni użytkownicy

Gdy wchodzisz na stronę internetową zawierającą treści, do których dostęp powinien być ograniczony, musisz zwykle podać odpowiedni **identyfikator** oraz **hasło**. Jeśli podana przez Ciebie kombinacja tych dwóch rzeczy jest poprawna, otrzymujesz dostęp, ponieważ się uwierzytelniłeś. Jeśli jesteś uwierzytelnionym użytkownikiem, system wie, że może Ci udostępnić wrażliwe treści. Obsługa tego stanu (posiadania uwierzytelnienia lub jego nieposiadania) wydaje się równie prosta, co ustawienie przełącznika na wartość `True` (dostęp przyznany; użytkownik zalogowany) lub `False` (dostęp zabroniony; użytkownik *nie*zalogowany).

Jest to nieco bardziej skomplikowane.

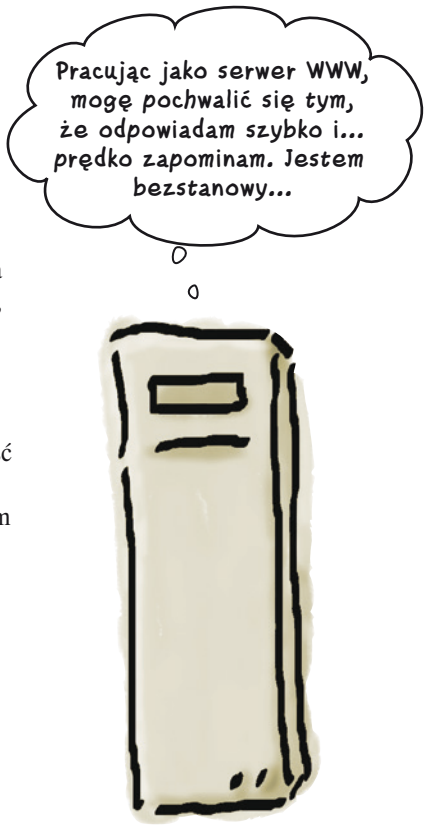
Z kwestią tą wiąże się trochę więcej problemów (dotyczących tego, jak działa internet), niż mogłoby się wydawać na pierwszy rzut oka. Przekonajmy się najpierw, na czym polegają te komplikacje (i sprawdźmy, jak sobie z nimi poradzić), zanim zajmiemy się rozwiązaniem kwestii ograniczenia dostępu do naszej strony.

Sieć jest bezstanowa

W swojej najprostszej postaci serwer sieciowy wydaje się niesamowicie „głupi”, ponieważ każde przetwarzane przez siebie żądanie sieciowe traktuje w całkowitym oderwaniu od pozostałych bez względu na to, czy nadeszły przed, czy też po nim.

Oznacza to, że szybkie wysłanie trzech żądań do serwera WWW z Twojego komputera będzie wyglądało jak trzy niezależne, *osobne* żądania. Nie ma tu znaczenia, że te trzy żądania wyszły z tej samej przeglądarki internetowej działającej na tym samym komputerze, który wykorzystuje ten sam, niezmieniający się adres IP (który serwer otrzymuje jako część składową żądania).

Jak już wspomnieliśmy na początku tej strony, wygląda tu na to, że serwer zachowuje się głupio. Mimo że przyjmujemy, iż te trzy żądania wysłane z Twojego komputera są ze sobą powiązane, serwer WWW nie postrzega tego w ten sposób: *każde żądanie sieciowe jest niezależne od tego, co przyszło przed nim, i tego, co przyjdzie po nim*.



Winę za to ponosi HTTP...

Powodem, dla którego serwery zachowują się w ten sposób, jest zastosowanie protokołu, na podstawie którego działa internet i który jest wykorzystywany zarówno przez serwer WWW, jak i przez Twoją przeglądarkę internetową. Protokołem tym jest HTTP (ang. *HyperText Transfer Protocol*, czyli „protokół przesyłania dokumentów hipertekstowych”).

HTTP stanowi, że serwery WWW mają działać tak, jak zostało to opisane powyżej, a powodem tego stanu rzeczy jest kwestia wydajności. Chodzi o to, aby zminimalizować ilość pracy wykonywanej przez serwery, dzięki czemu możliwe jest ich skalowanie w taki sposób, aby obsługiwały bardzo wiele żądań sieciowych. Zwiększenie wydajności odbywa się kosztem uwolnienia serwerów WWW od konieczności przechowywania informacji na temat serii żądań, które mogą być ze sobą powiązane. Informacja ta — znana w przypadku protokołu HTTP jako **stan** (i niezwiązana w żaden sposób z paradygmatem OOP) — nie obchodzi serwera WWW, ponieważ każde żądanie jest traktowane jako osobny byt. Dzięki temu serwer jest zoptymalizowany tak, aby szybko odpowiadać, lecz szybko również zapominać, a więc działać w sposób **bezstanowy**.

Wszystko to jest całkowicie w porządku i sprawdza się świetnie aż do chwili, gdy Twoja aplikacja WWW staje przed koniecznością zapamiętania czegoś.

Gdyby tylko działanie internetu było tak proste.

Gdy Twój kod działa jako element składowy serwera WWW, jego zachowanie może się różnić od tego, gdy działa na Twoim komputerze. Przyjrzyjmy się temu zagadnieniu uważnie.

Twój kod działa na serwerze WWW (nie na Twoim komputerze)

Gdy wykorzystująca framework Flask aplikacja WWW działa na Twoim komputerze, jej kod jest przez cały czas przechowywany w pamięci operacyjnej. Mając to na uwadze, przypomnij sobie ostatnie dwa wiersze należące do kodu Twojej aplikacji, o których była po raz pierwszy mowa w rozdziale 5. Wiersze te znajdziesz poniżej.

```
if __name__ == '__main__':
    app.run(debug=True)
```

Ten wiersz kodu NIE jest wykonywany, jeśli kod zostaje zaimportowany.

Przedstawiona powyżej instrukcja `if` sprawdza, czy interpreter wykonuje kod bezpośrednio, czy też kod jest importowany (przez interpreter lub jakiś inny mechanizm, taki jak *PythonAnywhere*). Gdy framework Flask działa na Twoim komputerze, kod Twojej aplikacji uruchamiany jest bezpośrednio, co powoduje, że wykonywana jest instrukcja `app.run`. Jeśli jednak do uruchomienia Twojej aplikacji wykorzystywany jest serwer WWW, jej kod jest *importowany*, a instrukcja `app.run` *nie* jest wykonywana.

Dlaczego tak jest? Ponieważ serwer WWW uruchamia kod Twojej aplikacji w taki sposób, *jaki uzna za stosowny*. Może to oznaczać, że zaimportuje kod aplikacji, a następnie będzie wywoływał jej funkcje wedle potrzeby, przez cały czas przechowując kod w pamięci. Może też jednak postanowić, że będzie ładował i usuwał kod aplikacji w zależności od potrzeb, zgodnie z założeniem, że w okresach braku aktywności ładowany i uruchamiany będzie tylko ten kod, który jest wymagany. W tym drugim przypadku — tj. gdy serwer ładuje kod tylko jeśli i tylko wtedy, kiedy jest to konieczne — mogą się zdarzyć problemy z zapisywaniem stanu aplikacji WWW w zmiennych. Zastanów się na przykład, co by było, gdybyś do kodu swojej aplikacji dodał przedstawiony poniżej wiersz kodu.

```
logged_in = False
if __name__ == '__main__':
    app.run(debug=True)
```

Ze zmiennej logged_in można by było korzystać do przechowywania informacji o tym, czy użytkownik Twojej aplikacji jest zalogowany, czy też nie.

Pomysł polega tu na tym, że w innych częściach aplikacji WWW można by było odwoływać się do zmiennej `logged_in`, aby sprawdzić, czy użytkownik jest autoryzowany. Poza tym w swoim kodzie mógłbyś zmieniać wartość tej zmiennej zgodnie z potrzebą (na przykład po udanej operacji logowania użytkownika). Ponieważ zmienna `logged_in` ma charakter *globalny*, można z niej korzystać i zmieniać jej wartość w całym kodzie Twojej aplikacji WWW. Podejście to wydaje się rozsądne, ale wiążą się z nim *dwa* problemy.

Po pierwsze, serwer WWW może w dowolnej chwili (i bez żadnego ostrzeżenia) usunąć wykonywany kod z pamięci, dlatego wszelkie wartości związane ze zmiennymi globalnymi mogą zostać **utracone** i przywrócone do wartości początkowych, gdy kod będzie ponownie importowany. Gdyby załadowana wcześniej funkcja ustawiła wartość zmiennej `logged_in` na `True`, a zaimportowany ponownie kod zresetował ją do wartości `False`, zapanowałby niezły chaos...

Nie przechowuj stanu swojej aplikacji WWW w zmiennych globalnych.

Po drugie, w Twoim działającym kodzie istnieje obecnie tylko *jedna kopia* zmiennej globalnej `logged_in`, co jest zupełnie w porządku, jeśli planujesz, aby Twoja aplikacja WWW już zawsze była wykorzystywana tylko przez jednego użytkownika naraz (w takim przypadku życzymy Ci powodzenia). Jeśli jednak zamierzasz, aby Twoja aplikacja miała dwóch lub więcej użytkowników, którzy odwołują się do tej zmiennej lub też zmieniają jej wartość, nie tylko zapanuje tu chaos, ale na gościnne występy wpadnie też frustracja. Jako ogólną zasadę przyjmij, że przechowywanie stanu aplikacji WWW w zmiennej globalnej jest bardzo złym pomysłem.

Czas na trochę sesji

Z informacji przedstawionych na poprzedniej stronie można wysnuć wniosek, że są nam potrzebne dwie rzeczy:

- sposób przechowywania danych bez uciekania się do używania zmiennych globalnych,
- sposób umożliwiający uniknięcie interferencji danych należących do różnych użytkowników aplikacji WWW.

Większość frameworków umożliwiających tworzenie aplikacji WWW (w tym Flask) zapewnia obydwie te rzeczy, oferując jedno rozwiązanie o nazwie **sesja**.

Sesję możesz traktować jako warstwę stanu rozproszoną nad bezstanowym internetem.

Framework Flask wykorzystuje technikę sesji do trzymania wszystkiego w ryzach przez dodanie niewielkiej porcji danych identyfikacyjnych do Twojej przeglądarki (w postaci *ciasteczka*) oraz połączenie ich z niewielką porcją danych identyfikacyjnych przechowywanych przez serwer WWW (w postaci *identyfikatora sesji*). Dzięki temu nie tylko możesz przechowywać stan swojej aplikacji WWW w taki sposób, aby był on zachowany mimo upływu czasu, lecz również zapewniasz, że każdy użytkownik Twojej aplikacji otrzymuje własną kopię tego stanu. Pozwala to wyeliminować chaos i frustrację.

Aby zademonstrować Ci działanie mechanizmu sesji zapewnianego przez framework Flask, przyjrzyjmy się niewielkiej aplikacji WWW, która została zapisana w pliku o nazwie *quick_session.py*. Poświęć chwilę na przeczytanie jej kodu, zwracając szczególną uwagę na wyróżnione fragmenty. Gdy to już zrobisz, przystąpimy do omówienia tego, co się dzieje w tym przykładzie.

Kod gotowy do użycia

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'NigdyNieZgadniesz'


@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'Użytkownikiem jest: ' + session['user']


@app.route('/getuser')
def getuser() -> str:
    return 'Użytkownikiem jest obecnie: ' + session['user']


if __name__ == '__main__':
    app.run(debug=True)
```

Pamiętaj, aby dodać session do listy importowanych elementów.

Twój tajny klucz będzie trudno odgadnąć.

To jest kod zapisany w pliku quick_session.py.

Danymi przechowywanymi w session możesz manipulować zgodnie z potrzebami.

Technologia sesji frameworka Flask zapewnia stan

Aby skorzystać z technologii sesji zapewnianej przez framework Flask, musisz najpierw zaimportować element `session` należący do modułu `flask`, co — jak widać — dzieje się już w pierwszym wierszu kodu aplikacji WWW zapisanej w pliku *quick_session.py*. O elemencie `session` możesz myśleć jako o globalnym słowniku Pythona, w którym przechowuje się stan swojej aplikacji WWW (aczkolwiek jest to słownik wzbogacony o pewne supermoce).

```
from flask import Flask, session
...
```

Zacznij od zaimportowania elementu session.

Mimo że Twoja aplikacja nada będzie działać w bezstanowym internecie, dzięki temu importowi będzie mogła zapamiętywać stan.

Framework Flask zapewnia, że wszelkie dane przechowywane w słowniku `session` istnieją przez cały czas działania aplikacji (niezależnie od tego, ile razy serwer WWW będzie ładował i przeładowywał kod aplikacji). Ponadto każda dana przechowywana w tym słowniku ma przypisany klucz utworzony przy użyciu unikatowego ciasteczka przeglądarki, co zapewnia, że dane Twojej sesji są odizolowane od danych wszystkich innych użytkowników aplikacji WWW.

Więcej informacji na temat sesji frameworka Flask znajdziesz pod adresem http://flask.pocoo.org/docs/0.11/api/#sessions.

Jak dokładnie framework Flask radzi sobie z tym wszystkim, nie jest dla nas istotne; liczy się tylko to, że jakoś to *robi*. Aby móc korzystać z tych wspaniałych możliwości, musisz uruchomić zapewniany przez Flaska mechanizm generowania ciasteczek, podając mu „tajny klucz", który jest używany przez framework do szyfrowania ciasteczek i zabezpiecza je przed wścibskim wzrokiem niepowołanych osób. Poniżej możesz zobaczyć, jak zostało to zrobione w programie *quick_session.py*.

```
...
app = Flask(__name__)

app.secret_key = 'NigdyNieZgadniesz'
...
```

Utwórz nową aplikację WWW Flaska w standardowy sposób.

Uruchom mechanizm generowania ciasteczek frameworka Flask, korzystając z tajnego klucza. (W tej roli sprawdzi się dowolny łańcuch znakowy, jednak — podobnie jak w przypadku wszystkich haseł, których używasz — zastosowany tu tekst powinno być trudno odgadnąć).

Dokumentacja frameworka Flask sugeruje wybranie takiego tajnego klucza, który trudno będzie odgadnąć, ale w praktyce można tu zastosować dowolny łańcuch znakowy. Flask używa tego klucza do szyfrowania Twojego ciasteczka przed przesłaniem go do przeglądarki.

Gdy element `session` jest już zaimportowany, a tajny klucz ustawiony, możesz w swoim kodzie skorzystać z `session` w taki sam sposób, w jaki korzystasz z dowolnego słownika w języku Python. W programie *quick_session.py* URL `/setuser` (i związana z nim funkcja `setuser`) przypisuje podaną przez użytkownika wartość do klucza `user` należącego do słownika `session`, a następnie zwraca tę wartość do przeglądarki, tak jak zostało to pokazane poniżej.

```
...
@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'Użytkownikiem jest: ' + session['user']
...
```

Wartość zmiennej user jest przypisywana do klucza 'user' w słowniku session.

URL powinien zawierać wartość, która ma być przypisana do zmiennej user (sposób działania tego mechanizmu poznasz już wkrótce).

Teraz, gdy mamy już ustawione pewne dane sesji, przyjrzyjmy się kodowi, za pomocą którego można się do nich odwołać.

Przeszukiwanie słownika umożliwia pobranie stanu

Kiedy wartość jest już powiązana z kluczem `user` w słowniku `session`, nie jest trudno odwołać się do niej, jeśli tylko zajdzie taka potrzeba.

Drugi URL używany w programie *quick_session.py*, `/getuser`, jest powiązany z funkcją o nazwie `getuser`. Wywołanie jej powoduje pobranie wartości związanej z kluczem `user` oraz zwrócenie jej oczekującej przeglądarce internetowej jako części komunikatu tekstowego. Funkcja `getuser` została przedstawiona poniżej wraz z kodem odpowiedzialnym za sprawdzenie warunku *dunder name równa się dunder main* (o którym po raz pierwszy była mowa w końcowej części rozdziału 5.).

```
...
@app.route('/getuser')
def getuser() -> str:
    return ' Użytkownikiem jest obecnie: ' + session['user']


if __name__ == '__main__':
    app.run(debug=True)
```

Odwoływanie się do danych zapisanych w słowniku session nie jest trudne. To zwykłe wyszukiwanie wartości w słowniku na podstawie określonego klucza.

Podobnie jak ma to miejsce w przypadku wszystkich aplikacji wykorzystujących framework Flask, za pomocą tego typowego dla Pythona kodu kontrolujemy, kiedy funkcja app.run zostaje uruchomiona.

Czas na jazdę próbną?

Nadszedł już niemal czas na wzięcie aplikacji WWW *quick_session.py* w obroty. Zanim to jednak zrobimy, zastanówmy się przez chwilę, co właściwie chcemy przetestować.

Na początek chcemy sprawdzić, czy aplikacja zapisuje i pobiera dane sesji, które są jej przekazywane. Ponadto zamierzamy się przekonać, czy może z niej korzystać więcej niż jeden użytkownik, nie depcząc innym osobom po palcach; chodzi w końcu o to, że dane sesji jednego użytkownika nie powinny mieć wpływu na dane żadnego innego.

Aby wykonać wspomniane tu testy, zamierzamy zasymulować obecność wielu użytkowników, uruchamiając wiele przeglądarek internetowych. Choć wszystkie one będą działać na jednym komputerze, z punktu widzenia serwera WWW każda z nich będzie utrzymywać własne, niezależne połączenie; jakby nie było, internet jest przecież bezstanowy. Gdybyśmy przeprowadzili te testy na trzech osobnych, fizycznie różnych komputerach używających trzech różnych sieci, wynik byłby taki sam, ponieważ wszystkie serwery sieciowe rozpatrują każde żądanie w izolacji, niezależnie od tego, skąd te żądania pochodzą. Pamiętaj, że rozwiązanie zapewniane przez framework Flask za pomocą słownika `session` daje stanowy charakter bezstanowemu internetowi.

Aby uruchomić tę aplikację WWW, skorzystaj z przedstawionego poniżej polecenia w systemach *Linux* lub *Mac OS X*.

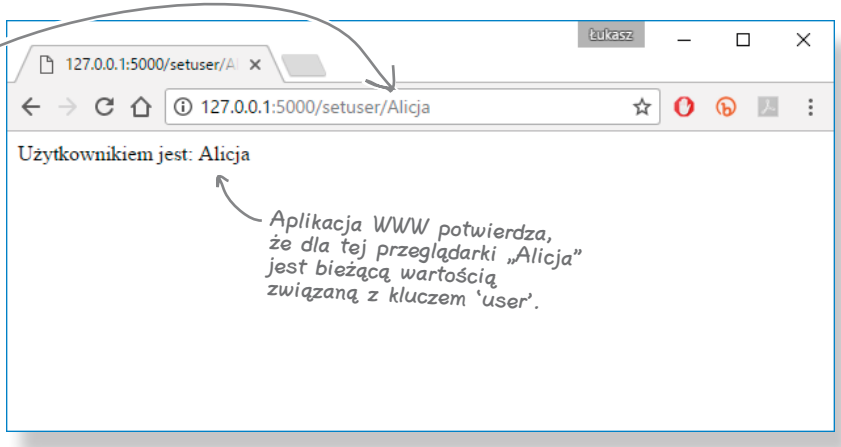
```
$ python3 quick_session.py
```

Jeśli zaś używasz systemu *Windows*, posłuż się następującym poleceniem:

```
C:\> py -3 quick_session.py
```

Jazda próbna

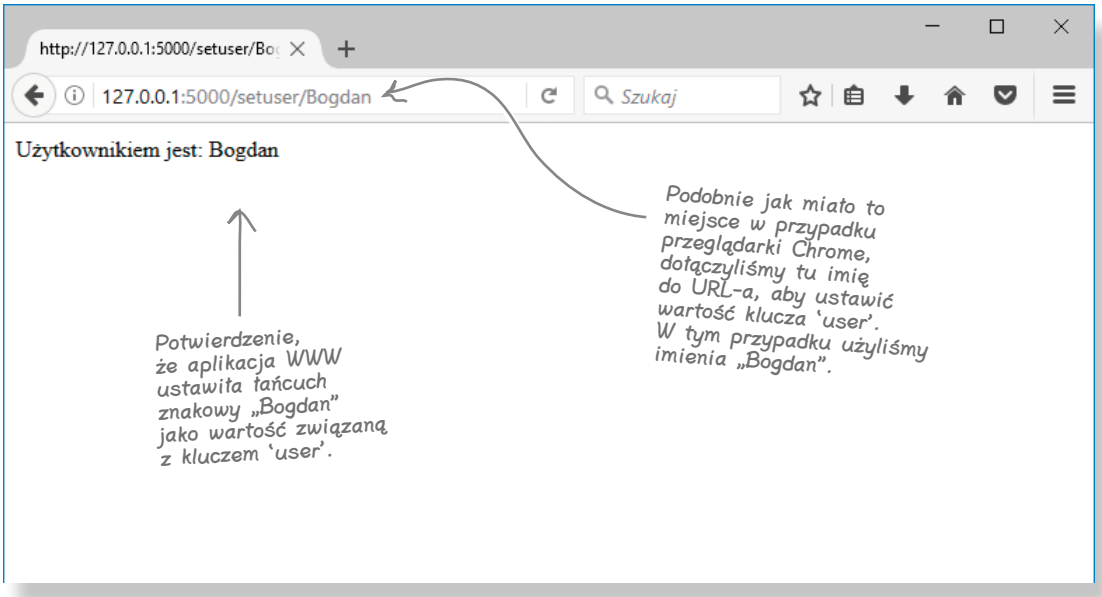
Uruchomiwszy aplikację WWW *quick_session.py*, otwórzmy okno przeglądarki *Chrome* i użyjmy go, aby ustawić wartość przypisaną do klucza `user` w słowniku `session`. Zrobimy to, wpisując w pasku adresu ciąg **`/setuser/Alicja`**, co stanowi instrukcję dla aplikacji, że ma przypisać kluczowi `user` wartość `Alicja`, tak jak zostało to przedstawione poniżej.



Dodanie imienia do końca ciągu URL informuje aplikację WWW, że ma użyć słowa „Alicja" jako wartości dla klucza 'user'.

Aplikacja WWW potwierdza, że dla tej przeglądarki „Alicja" jest bieżącą wartością związaną z kluczem 'user'.

Następnie otwórz przeglądarkę *Firefox* i użyj jej do przypisania kluczowi `user` wartości Bogdan (jeśli nie masz dostępu do tego programu, skorzystaj z dowolnej przeglądarki, jaką masz pod ręką, byle tylko nie była to aplikacja *Chrome*).

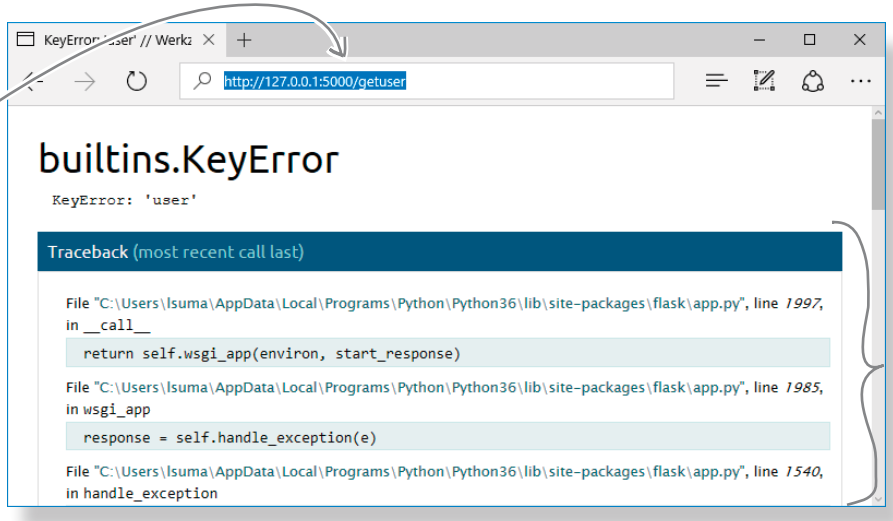


Potwierdzenie, że aplikacja WWW ustawiła łańcuch znakowy „Bogdan" jako wartość związaną z kluczem 'user'.

Podobnie jak miało to miejsce w przypadku przeglądarki Chrome, dołączyliśmy tu imię do URL-a, aby ustawić wartość klucza 'user'. W tym przypadku użyliśmy imienia „Bogdan".

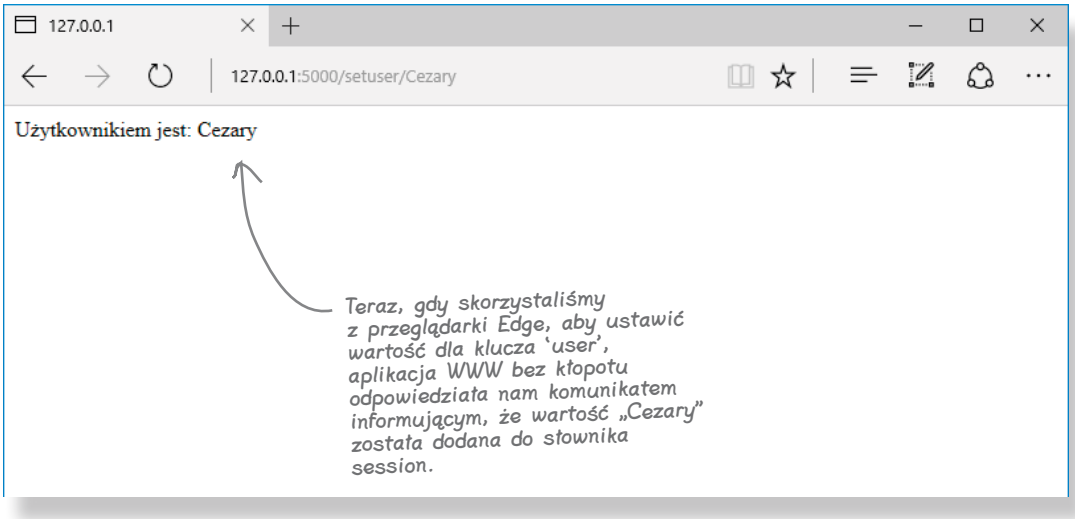
Gdy otworzyliśmy przeglądarkę *Edge* (lub *Safari*, gdybyś korzystał z systemu *Mac OS X*), użyliśmy innego URL-a obsługiwanego przez naszą aplikację WWW, a mianowicie ciągu `/getuser`, aby otrzymać bieżącą wartość związaną z kluczem `user`. Gdy tylko to jednak zrobiliśmy, zobaczyliśmy dość przerażający komunikat błędu, który został pokazany poniżej.

ULR /getuser umożliwia sprawdzenie bieżącej wartości powiązanej z kluczem 'user'.



Ojej! To wygląda na komunikat błędu, czyż nie? Najważniejsza informacja znajduje się na samej górze: mamy tu do czynienia z błędem KeyError, ponieważ nie użyliśmy wcześniej przeglądarki Edge, aby ustawić wartość powiązaną z kluczem 'user'. (Przypomnij sobie, że ustawialiśmy ją za pomocą programów Chrome i Firefox, nie za pomocą Edge'a).

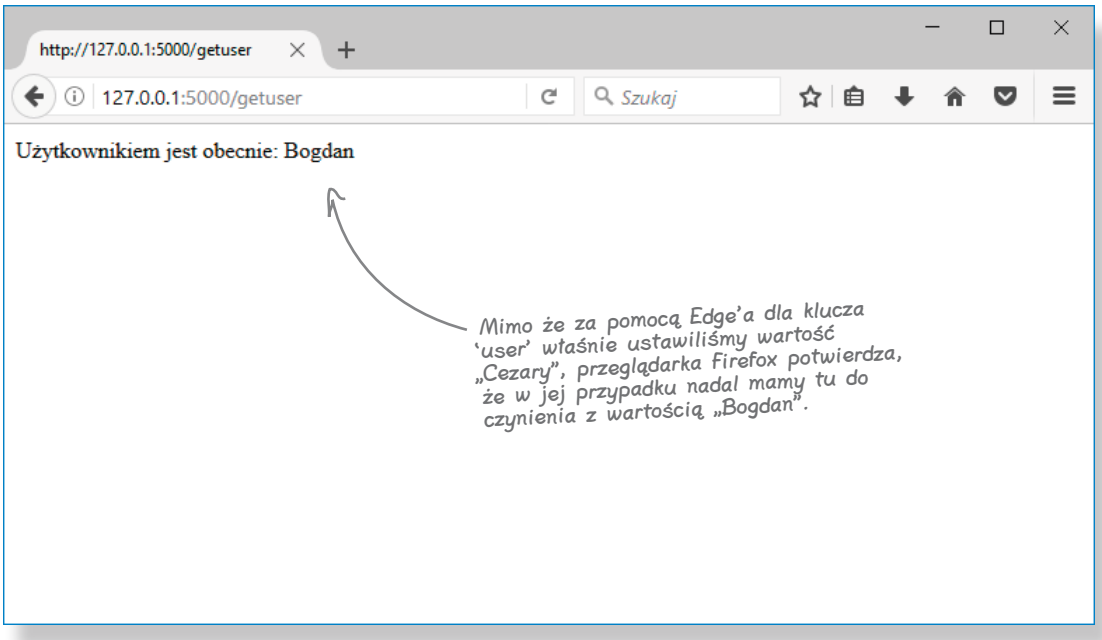
Skorzystajmy z przeglądarki *Edge*, aby przypisać łańcuch znakowy `Cezary` do wartości związanej z kluczem `user`.



Teraz, gdy skorzystaliśmy z przeglądarki Edge, aby ustawić wartość dla klucza 'user', aplikacja WWW bez kłopotu odpowiedziała nam komunikatem informującym, że wartość „Cezary" została dodana do słownika session.

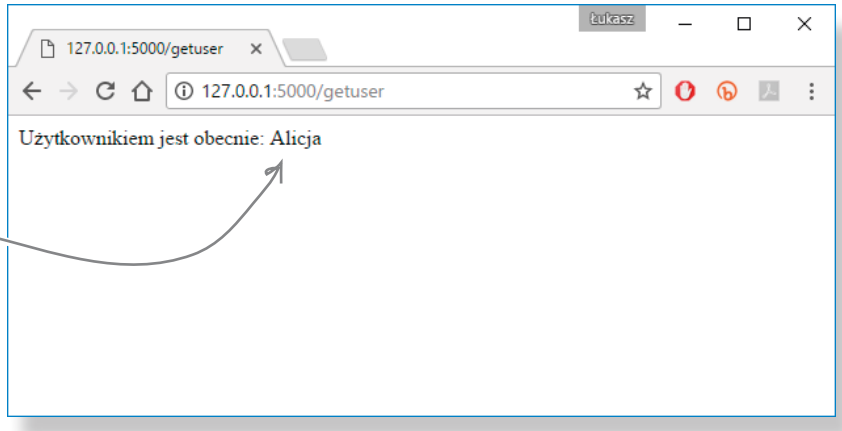
Jazda próbna

Gdy użyliśmy już trzech różnych przeglądarek internetowych do ustawienia wartości związanej z kluczem `user`, potwierdźmy, że nasza aplikacja WWW (dzięki zastosowaniu słownika `session`) uniemożliwia interferencję wartości określonych za pomocą poszczególnych programów. Mimo że przed chwilą przypisaliśmy kluczowi `user` wartość `Cezary`, korzystając z przeglądarki *Edge*, program *Firefox* pokaże nam inne dane, gdy użyjemy go do przejścia pod URL `/getuser`, tak jak zostało to pokazane poniżej.



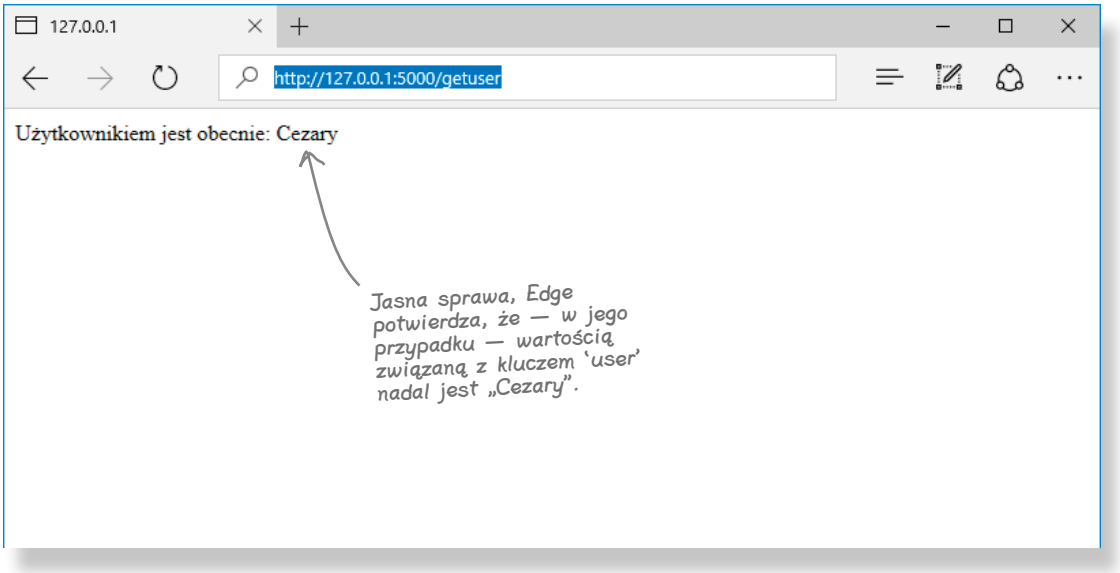
Mimo że za pomocą Edge'a dla klucza 'user' właśnie ustawiliśmy wartość „Cezary", przeglądarka Firefox potwierdza, że w jej przypadku nadal mamy tu do czynienia z wartością „Bogdan".

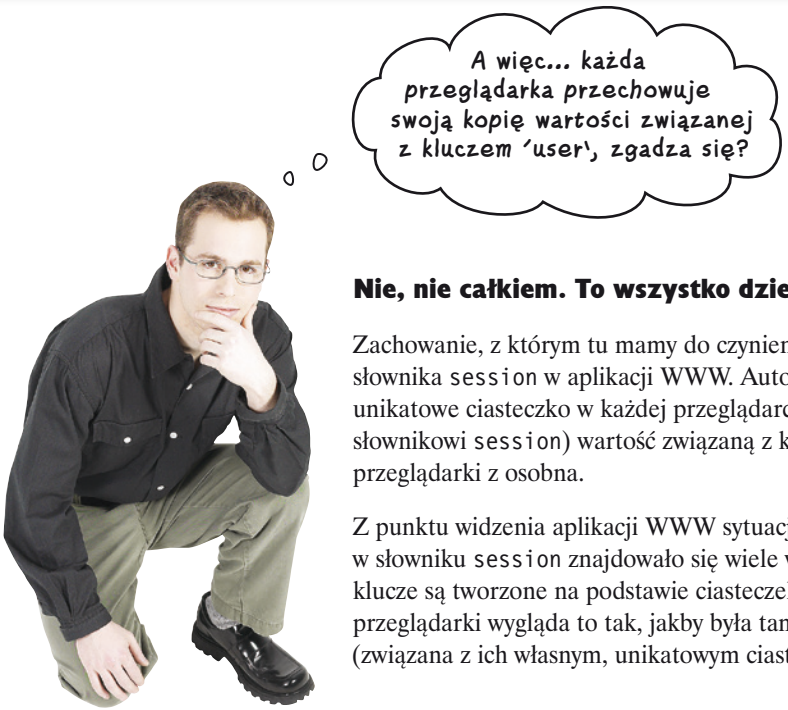
Potwierdziwszy, że *Firefox* pokazuje dla klucza `user` wartość `Bogdan`, powróćmy do okna przeglądarki *Chrome* i przejdźmy w nim pod URL `/getuser`. Tak jak można się było spodziewać, *Chrome* potwierdza, że w jego przypadku wartością związaną z kluczem `user` jest `Alicja`.



Zgodnie z przewidywaniami Chrome nadal uważa, że wartością związaną z kluczem 'user' jest „Alicja".

Właśnie skorzystaliśmy z przeglądarek *Firefox* i *Chrome*, aby uzyskać dostęp do wartości związanej z kluczem `user`, używając w tym celu URL-a `/getuser`, co oznacza, że został nam do sprawdzenia już tylko program *Edge*. Poniżej możesz się przekonać, co zobaczyliśmy, gdy za jego pomocą przeszliśmy pod ten URL. Tym razem nie otrzymaliśmy komunikatu błędu, ponieważ z kluczem `user` związana jest już wartość (nie występuje więc już tutaj `KeyError`).





Nie, nie całkiem. To wszystko dzieje się w aplikacji WWW.

Zachowanie, z którym tu mamy do czynienia, zawdzięczamy użyciu słownika `session` w aplikacji WWW. Automatycznie ustawiając unikatowe ciasteczko w każdej przeglądarce, posiada ona (dzięki słownikowi `session`) wartość związaną z kluczem `user` dla *każdej* przeglądarki z osobna.

Z punktu widzenia aplikacji WWW sytuacja przedstawia się tak, jakby w słowniku `session` znajdowało się wiele wartości `user` (których klucze są tworzone na podstawie ciasteczek). Z perspektywy każdej przeglądarki wygląda to tak, jakby była tam tylko jedna wartość `user` (związana z ich własnym, unikatowym ciasteczkiem).

Zarządzanie logowaniem za pomocą sesji

Dzięki naszym doświadczeniom z programem *quick_session.py* wiemy, że w słowniku `session` możemy przechowywać stan specyficzny dla przeglądarki. Niezależnie od tego, ile przeglądarek łączy się z Twoją aplikacją WWW, związane z każdą z nich dane przechowywane po stronie serwera (czyli *stan*) są zarządzane przez framework Flask zawsze wtedy, gdy używamy słownika `session`.

Skorzystajmy z tej świeżo nabytej wiedzy, aby powrócić do problemu kontrolowania dostępu do określonych stron należących do aplikacji WWW *vsearch4web.py*. Przypomnij sobie, że chcemy dotrzeć do punktu, w którym będziemy mogli wskazać, kto ma dostęp do strony znajdującej się pod URL-em `/viewlog`.

Zamiast eksperymentować z naszym działającym kodem programu *vsearch4web.py*, odstawmy go na razie na bok i zajmijmy się innym, z którym poeksperymentujemy, aby dowiedzieć się, co powinniśmy zrobić. Do kodu programu *vsearch4web.py* wrócimy, gdy tylko uda nam się wypracować najlepszy sposób rozwiązania naszego problemu. Będziemy wtedy mogli spokojnie zmienić kod naszej aplikacji WWW tak, aby ograniczyć dostęp do strony znajdującej się pod URL-em `/viewlog`.

Poniżej znajdziesz kod jeszcze jednej aplikacji wykorzystującej framework Flask. Podobnie jak wcześniej, poświęć trochę czasu, aby przeczytać ten kod, zanim przejdziesz do lektury jego opisu. Program zapisany jest w pliku *simple_webapp.py*.

Kod gotowy do użycia

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Witaj, świecie, tu prosta aplikacja WWW.'

@app.route('/page1')
def page1():
    return 'To jest strona  1.'

@app.route('/page2')
def page2():
    return 'To jest strona  2.'

@app.route('/page3')
def page3():
    return 'To jest strona  3.'

if __name__ == '__main__':
    app.run(debug=True)
```

To jest kod pochodzący z pliku simple_webapp.py. Na tym etapie nie powinieneś mieć problemu z przeczytaniem go i zrozumieniem sposobu działania tej aplikacji WWW.

Zajmijmy się logowaniem

Kod programu *simple_webapp.py* jest bardzo prosty: wszystkie znajdujące się w nim URL-e są publiczne w tym sensie, że za pomocą przeglądarki internetowej może się do nich odwoływać dosłownie każda osoba.

Oprócz standardowego URL-a / (którego użycie powoduje wykonanie funkcji `hello`) znajdują się tam trzy inne ciągi: `/page1`, `/page2` oraz `/page3` (których użycie wywołuje funkcje o podobnych nazwach). Skorzystanie z każdego z URL-i powoduje zwrócenie przeglądarce odpowiedniego komunikatu.

Nasza niewielka aplikacja to tak naprawdę tylko skorupa, ale wystarczy do naszych celów. Chcielibyśmy osiągnąć punkt, w którym strony znajdujące się pod URL-ami `/page1`, `/page2` oraz `/page3` będą widoczne wyłącznie dla zalogowanych użytkowników, niedostępne zaś dla wszystkich innych. Aby to umożliwić, mamy zamiar skorzystać z możliwości oferowanych przez słownik `session` dostarczany przez framework Flask.

Zacznijmy od obsługi naprawdę prostego mechanizmu logowania dostępnego pod URL-em `/login`. Na tym etapie nie będziemy się przejmowali zapewnieniem formularza HTML umożliwiającego użytkownikowi podanie identyfikatora logowania i hasła. Wszystko, co zamierzamy tu zrobić, sprowadza się do utworzenia kodu, który zapisuje w słowniku `session` informację, że użytkownikowi udało się prawidłowo zalogować.

Zaostrz ołówek

Poniżej napisz kod obsługi URL-a `/login`. W pustych miejscach wpisz instrukcje, za pomocą których zmienisz słownik `session`, ustawiając dla klucza `logged_in` wartość `True`. Spraw ponadto, aby związana z tym URL-em funkcja zwracała do oczekującej przeglądarki komunikat „Teraz jesteś zalogowany".

```
@app.route('/login')
def do_login() -> str:
    ..............................................................
    return ........................................................
```

Tutaj dodaj nowy kod.

Oprócz uzupełnienia kodu obsługi URL-a `/login` powinieneś dokonać w nim jeszcze dwóch innych zmian mających na celu włączenie mechanizmu sesji. Opisz szczegółowo, z czym Twoim zdaniem wiążą się te modyfikacje.

1. .....

2. .....

Zaostrz ołówek
Rozwiązanie

Poniżej miałeś napisać kod obsługi URL-a `/login`. W pustych miejscach miałeś wpisać instrukcje, za pomocą których można zmienić słownik `session`, ustawiając dla klucza `logged_in` wartość `True`. Miałeś ponadto sprawić, aby związana z tym URL-em funkcja zwracała do oczekującej przeglądarki komunikat „Teraz jesteś zalogowany".

```
@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'Teraz jesteś zalogowany'
```

Ustaw wartość True dla klucza 'logged_in' należącego do słownika session.

Zwróć ten komunikat do oczekującej przeglądarki.

Oprócz uzupełnienia kodu obsługi URL-a `/login` powinieneś dokonać w nim jeszcze dwóch innych zmian mających na celu włączenie mechanizmu sesji. Miałeś szczegółowo opisać, z czym Twoim zdaniem wiążą się te modyfikacje.

1. Musimy dodać element session do wiersza importowania znajdującego się na początku kodu.
2. Musimy ustawić wartość tajnego klucza dla tej aplikacji WWW.

Nie zapomnijmy o zrobieniu tych dwóch rzeczy.

Zmień kod aplikacji WWW w taki sposób, aby obsługiwała logowanie

Z testowaniem nowego kodu wstrzymamy się do czasu, gdy dodamy dwa kolejne URL-e, `/logout` i `/status`, oraz odpowiedni kod ich obsługi. Zanim do tego przejdziemy, upewnij się, że Twoja kopia pliku *simple_webapp.py* została zmieniona tak, aby uwzględniać modyfikacje przedstawione poniżej. Zauważ, że nie pokazujemy tu całego kodu aplikacji WWW, a jedynie jego nowe fragmenty (które zostały wyróżnione).

```
from flask import Flask, session

app = Flask(__name__)

    ...

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'Teraz jesteś zalogowany.'

app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

Pamiętaj, aby zaimportować element session.

Dodaj kod obsługi URL-a /login.

Ustaw wartość tajnego klucza dla tej aplikacji WWW (umożliwia nam to użycie sesji).

Zajmijmy się wylogowywaniem i sprawdzaniem stanu

Naszym następnym zadaniem jest dodanie kodu obsługi URL-i `/logout` i `/status`.

Jeśli chodzi o operację wylogowywania, jedną ze strategii może tu być ustawienie wartości związanej z kluczem `logged_in` słownika `session` na `False`. Inną strategią jest całkowite *usunięcie* tego klucza ze słownika. Wybierzemy drugą opcję. Powód, dla którego tak zrobimy, zostanie wyjaśniony, gdy opracujemy kod obsługi URL-a `/status`.

Zaostrz ołówek

Napisz kod obsługi URL-a `/logout`, za pomocą którego należy usunąć klucz `logged_in` ze słownika `session`, a następnie zwrócić do oczekującej przeglądarki komunikat o treści „Teraz jesteś wylogowany".

Wskazówka: jeśli zdążyłeś już zapomnieć, jak usuwa się klucz ze słownika, wpisz po znaku zachęty >>> polecenie dir(dict), aby wyświetlić listę metod dostępnych dla słownika.

Tutaj dodaj kod odpowiedzialny za operację wylogowania.

```
@app.route('/logout')
def do_logout() -> str:
    ..........................................................
    return ..................................................
```

Mając już napisany kod obsługi URL-a `/logout`, możemy skupić się na obsłudze URL-a `/status`. Polega ona na zwróceniu do oczekującej przeglądarki internetowej jednego z dwóch komunikatów.

Komunikat o treści „W tej chwili jesteś zalogowany" ma być zwracany, gdy klucz `logged_in` istnieje jako wartość w słowniku `session` (i z definicji jest ustawiona na `True`).

Komunikat o treści „NIE jesteś zalogowany" będzie zwracany, gdy w słowniku `session` nie ma klucza `logged_in`. Zauważ, że nie możemy tu sprawdzać, czy klucz ten ma wartość `False`, ponieważ kod uruchamiany w wyniku wywołania URL-a `/logout` usuwa go ze słownika, a nie jedynie zmienia jego wartość. (Nie zapomnieliśmy, że nadal musimy wyjaśnić, dlaczego robimy to w ten sposób, i wkrótce się tym zajmiemy. Na razie zaufaj nam po prostu, że właśnie tak należy zaimplementować tę funkcję).

Napisz kod obsługi URL-a `/status` w pustych miejscach widocznych poniżej.

Sprawdź, czy klucz 'logged_in' istnieje w słowniku session, a następnie zwróć odpowiedni komunikat.

Tutaj umieść swój kod odpowiedzialny za sprawdzanie stanu.

```
@app.route('/status')
def check_status() -> str:
    if ..............................................
        return ......................................
    return ..........................................
```

Zaostrz ołówek
Rozwiązanie

Twoim zadaniem było napisanie kodu obsługi URL-a `/logout`, za pomocą którego należy usunąć klucz `logged_in` ze słownika `session`, a następnie zwrócić do oczekującej przeglądarki komunikat o treści „Teraz jesteś wylogowany".

```
@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'Teraz jesteś wylogowany.'
```

Użyj metody pop, aby usunąć klucz 'logged_in' ze słownika session.

Mając już napisany kod obsługi URL-a `/logout`, miałeś się skupić na obsłudze URL-a `/status`. Polega ona na zwróceniu do oczekującej przeglądarki internetowej jednego z dwóch komunikatów.

Komunikat o treści „W tej chwili jesteś zalogowany" miał być zwracany, gdy klucz `logged_in` istnieje jako wartość w słowniku `session` (i z definicji jest ustawiona na `True`).

Komunikat o treści „NIE jesteś zalogowany" miał być zwracany, gdy w słowniku `session` nie ma klucza `logged_in`.

Miałeś napisać kod obsługi URL-a `/status` w pustych miejscach widocznych poniżej.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'W tej chwili jesteś zalogowany.'
    return 'NIE jesteś zalogowany.'
```

Czy w słowniku session znajduje się klucz 'logged_in'?

Jeśli tak, zwróć ten komunikat.

Jeśli nie, zwróć ten komunikat.

Zmień kod aplikacji WWW raz jeszcze

Nadal wstrzymujemy się z testowaniem tej nowej wersji aplikacji WWW, ale tutaj (po prawej stronie) wyróżniliśmy fragmenty kodu, które powinieneś dodać do swojej kopii pliku *simple_webapp.py*.

Upewnij się, że zmieniłeś swój kod w taki sposób, aby wyglądał tak samo jak nasz, zanim przejdziesz do kolejnej *Jazdy próbnej*, która nastąpi tuż po tym, gdy spełnimy naszą wcześniejszą obietnicę.

```
...
@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'Teraz jesteś wylogowany.'

@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'W tej chwili jesteś zalogowany.'
    return 'NIE jesteś zalogowany.'

app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

Dwie nowe funkcje obsługi ciągów URL

Dlaczego nie sprawdzać wartości False?

Opracowując kod obsługi URL-a `/login`, ustawiłeś wartość związaną z należącym do słownika `session` kluczem `logged_in` na `True` (co wskazywało, że użytkownik przeglądarki był zalogowany w aplikacji WWW). Jednak tworząc kod obsługi URL-a `/logout`, nie ustawiałeś wartości związanej z tym kluczem na `False`, ponieważ wybraliśmy zamiast tego rozwiązanie polegające na usunięciu klucza `logged_in` ze słownika. W kodzie obsługi URL-a `/status` sprawdzaliśmy „stan logowania", określając, czy klucz `logged_in` istnieje w słowniku `session`; nie sprawdzaliśmy, czy związaną z nim wartością jest `False` (czy też `True`). Prowadzi to do pytania: *dlaczego aplikacja WWW nie korzysta z wartości `False`, aby wskazywać stan „niezalogowany"?*

Odpowiedź może być na początku dość trudna do zrozumienia, ale jest ważna i wiąże się ze sposobem, w jaki w języku Python działają słowniki. Aby zilustrować ten problem, wykonajmy kilka eksperymentów w wierszu poleceń powłoki Pythona i zasymulujmy, co dzieje się ze słownikiem `session`, gdy jest on używany w aplikacji WWW. Staraj się robić to, co my, i uważnie przeczytać każdy z przedstawionych poniżej komentarzy.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> session = dict()
>>>
>>> if session['logged_in']:
        print('Znaleziony.')


Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    if session['logged_in']:
KeyError: 'logged_in'
>>>
>>> if 'logged_in' in session:
        print('Znaleziony.')


>>> session['logged_in'] = True
>>>
>>> if 'logged_in' in session:
        print('Znaleziony.')


Znaleziony.
>>>
>>> if session['logged_in']:
        print('Znaleziony.')


Znaleziony.
>>>
                                  Ln: 43  Col: 4
```

Utwórz nowy, pusty słownik o nazwie „session".

Spróbuj sprawdzić, czy istnieje w nim klucz 'logged_in', korzystając z instrukcji if.

Ups! Klucza 'logged_in' jeszcze w nim nie ma, dlatego pojawia się błąd KeyError, w wyniku czego nasz kod ulega awarii.

Jeśli jednak sprawdzamy obecność klucza w słowniku za pomocą operatora in, nasz kod nie ulega awarii (nie ma błędu KeyError), mimo że do klucza nie jest przypisana żadna wartość.

Przypiszmy kluczowi 'logged_in' jakąś wartość.

Sprawdzanie obecności klucza w słowniku za pomocą operatora in w dalszym ciągu działa, choć tym razem wynik jest pozytywny (jako że klucz istnieje i jest mu przypisana pewna wartość).

Sprawdzanie obecności klucza w słowniku za pomocą instrukcji if również działa (teraz, gdy z kluczem jest już związana pewna wartość). Gdyby jednak klucz ten został usunięty ze słownika (za pomocą metody pop), kod ten ponownie powodowałby wystąpienie błędu KeyError.

Przedstawione powyżej eksperymenty pokazują, że **nie** da się sprawdzać wartości związanej z kluczem należącym do słownika, dopóki odpowiednia para klucz-wartość nie istnieje. Podjęcie takiej próby powoduje wystąpienie błędu `KeyError`. Jako że lepiej unikać błędów tego rodzaju, w kodzie programu *simple_webapp.py* sprawdzamy istnienie klucza `logged_in`, aby przekonać się, czy użytkownik przeglądarki jest zalogowany, zamiast sprawdzać rzeczywistą wartość tego klucza. Pozwala nam to uniknąć ryzyka pojawienia się błędu `KeyError`.

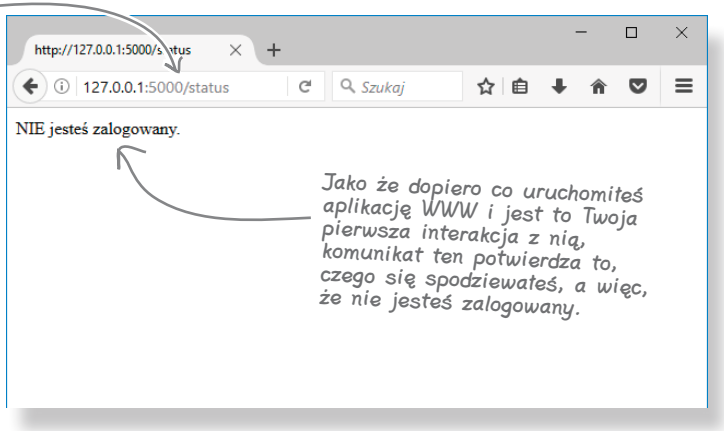
Jazda próbna

Weźmy naszą prostą aplikację *simple_webapp.py* w obroty, aby przekonać się, jak dobrze sprawuje się kod obsługi URL-i `/login`, `/logout` i `/status`. Podobnie jak miało to miejsce w przypadku naszej poprzedniej *Jazdy próbnej*, mamy zamiar przetestować tę nową aplikację WWW, korzystając z różnych przeglądarek internetowych, aby potwierdzić, że każda z nich posiada własny „stan zalogowania" na serwerze. Uruchommy aplikację z poziomu terminala naszego systemu operacyjnego w sposób przedstawiony poniżej.

- W systemach *Linux* oraz *Mac OS X* wydaj polecenie: `python3 simple_webapp.py`.
- W systemie *Windows* użyj polecenia: `py -3 simple_webapp.py`.

Uruchommy przeglądarkę *Firefox* i sprawdźmy początkowy stan zalogowania, odwołując się do URL-a `/status`. Zgodnie z oczekiwaniami użytkownik przeglądarki nie jest zalogowany, co widać poniżej.

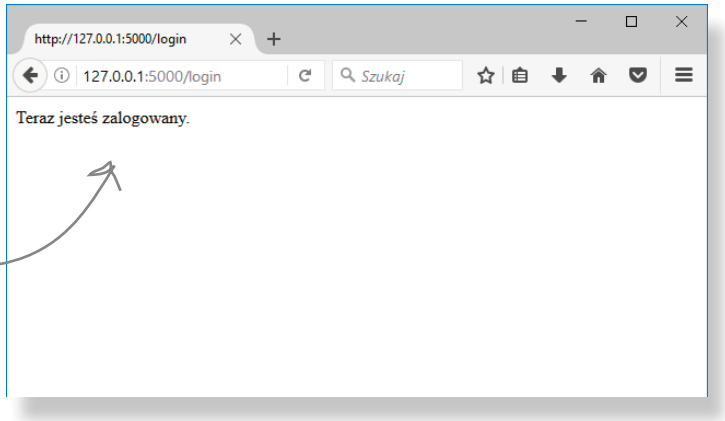
Przejdź do URL-a /status, aby określić, czy użytkownik przeglądarki jest zalogowany, czy też nie.



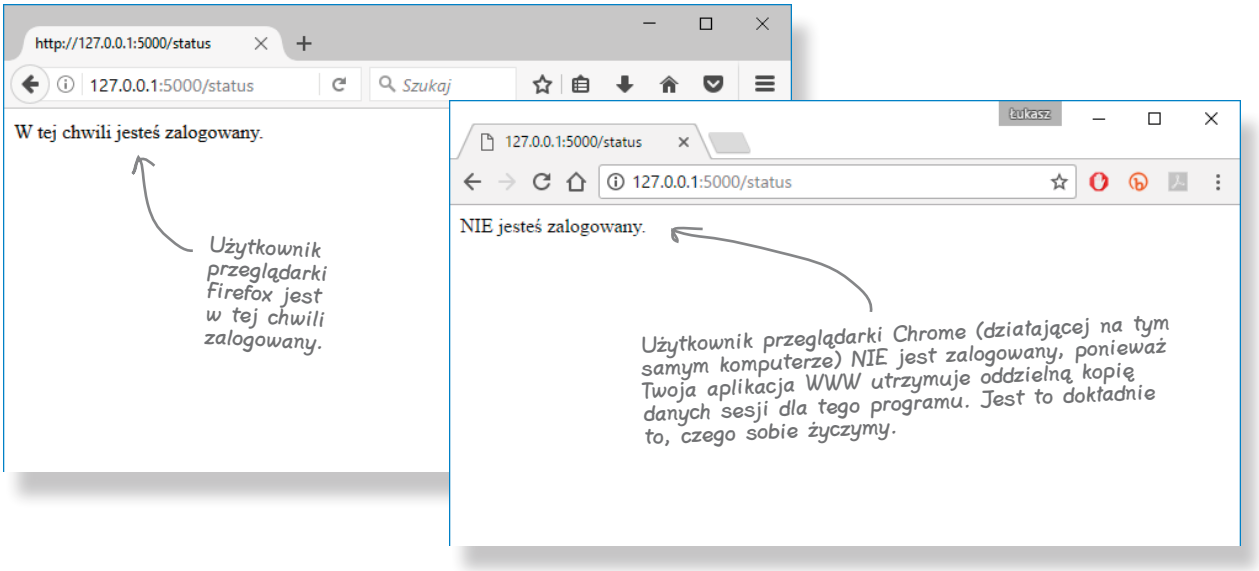
Jako że dopiero co uruchomiłeś aplikację WWW i jest to Twoja pierwsza interakcja z nią, komunikat ten potwierdza to, czego się spodziewałeś, a więc, że nie jesteś zalogowany.

Zasymulujmy operację logowania, przechodząc do URL-a `/login`. Komunikat zmieni się, potwierdzając, że logowanie się udało.

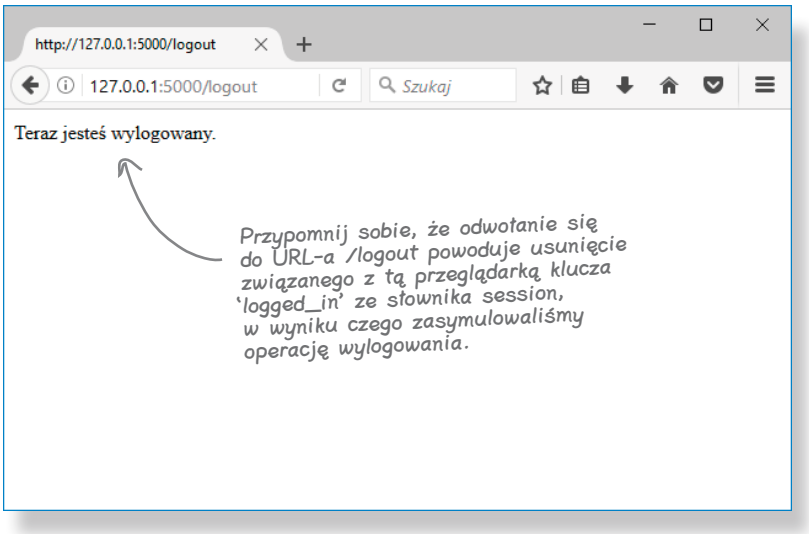
Przejście do URL-a /login powoduje dokładnie to, czego się spodziewamy. Użytkownik przeglądarki jest teraz zalogowany w aplikacji WWW.



Teraz, gdy jesteś już zalogowany, potwierdźmy zmianę stanu, odwołując się do URL-a `/status` za pomocą przeglądarki *Firefox*. Gdy sprawdzisz stan również za pomocą programu *Chrome*, przekonasz się, że użytkownik tej przeglądarki nie jest zalogowany, a więc jest dokładnie tak, jak chcemy (jako że każdy użytkownik aplikacji WWW — w tym przypadku każda przeglądarka — ma własny stan utrzymywany przez tę aplikację), co widać poniżej.



Na koniec sprawdźmy URL `/logout` za pomocą programu *Firefox*, aby poinformować aplikację WWW, że wylogowujemy się z naszej sesji, tak jak zostało to pokazane poniżej.



Mimo że nie prosiliśmy użytkowników naszej przeglądarki o podanie identyfikatora czy hasła, URL-e `/login`, `/logout` i `/status` umożliwiły nam zasymulowanie tego, co działoby się ze słownikiem `session` aplikacji WWW, gdybyśmy opracowali odpowiedni formularz HTML, a następnie połączyli wprowadzone za jego pomocą dane z bazą danych dostępowych utrzymywaną po stronie backendu. Szczegóły związane z tym, jak dokładnie miałoby się to odbywać, są w dużej mierze zależne od konkretnej aplikacji, ale podstawowy mechanizm (tj. manipulowanie słownikiem `session`) pozostaje taki sam niezależnie od tego, co aplikacja ta miałaby robić.

Czy jesteśmy już gotowi, aby ograniczyć dostęp do stron związanych z URL-ami `/page1`, `/page2` oraz `/page3`?

Czy możemy teraz ograniczyć dostęp do URL-i?

Kuba: Hej, Franek... Z czym masz problem?

Franek: Muszę wymyślić jakiś sposób, żeby ograniczyć dostęp do stron związanych z URL-ami `/page1`, `/page2` oraz `/page3`...

Józek: To nie może być zbyt trudne, prawda? W funkcji obsługującej URL `/status` masz już przecież kod, którego potrzebujesz...

Franek: ...i wiadomo dzięki niemu, czy użytkownik przeglądarki jest zalogowany, czy nie, zgadza się?

Józek: Tak, dokładnie. Zatem musisz jedynie skopiować ten kod sprawdzający stan zalogowania z funkcji obsługującej URL `/status` i wkleić go do każdej funkcji obsługi URL-i, do których dostęp chcesz ograniczyć. Potem będziesz już w domu!

Kuba: Człowieku! Kopiowanie i wklejanie to *pięta achillesowa* programistów aplikacji WWW. Naprawdę nie powinieneś kopiować i wklejać kodu w ten sposób... Może to tylko doprowadzić do problemów, które pojawią się prędzej czy później.

Franek: Oczywiście! Podstawy informatyki... W takim razie utworzę funkcję zawierającą kod używany do obsługi URL-a `/status`, a następnie wywołam *tę* funkcję, gdy będzie to potrzebne wewnątrz funkcji obsługujących URL-e `/page1`, `/page2` oraz `/page3`. Problem z głowy.

Józek: Ten pomysł mi się podoba... i myślę, że to rozwiązanie się sprawdzi. (Wiedziałem, że siedzenie na tych wszystkich *nudnych* wykładach z podstaw informatyki w końcu się na coś przyda).

Kuba: Poczekajcie... nie tak szybko. Twoja sugestia dotycząca funkcji jest znacznie lepsza niż pomysł z kopiowaniem i wklejaniem kodu, ale nadal nie jestem przekonany, czy jest to najlepsze rozwiązanie.

Franek i **Józek** (razem i z niedowierzaniem): *Co Ci się tu nie podoba?!?!?!*

Kuba: Irytuje mnie, że do funkcji obsługi URL-i `/page1`, `/page2` oraz `/page3` planujecie dodać kod, który nie ma nic wspólnego z tym, co naprawdę *robią* te funkcje. Zgoda, musisz sprawdzić, czy użytkownik jest zalogowany, zanim dasz mu dostęp do odpowiednich stron, ale dodanie wywołania funkcji w każdym kodzie obsługi URL-i nie wydaje mi się dobrym rozwiązaniem...

Franek: Jaki zatem masz wspaniały pomysł?

Kuba: Gdyby to ode mnie zależało, utworzyłbym tu dekorator, a następnie z niego skorzystał.

Józek: Oczywiście! To nawet lepszy pomysł! Zróbmy to.

Kopiowanie i wklejanie rzadko okazuje się dobrym pomysłem

Przekonajmy się, że pomysły zaprezentowane na poprzedniej stronie *nie* są najlepszymi sposobami rozwiązania problemu, z którym mamy do czynienia, a konkretnie tego, jak poradzić sobie z ograniczeniem dostępu do wybranych stron WWW.

Pierwszą sugestią było skopiowanie i wklejenie pewnego fragmentu kodu odpowiedzialnego za obsługę URL-a `/status` (chodziło tu o funkcję `check_status`). Poniżej możesz zobaczyć kod, o którym mowa.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'W tej chwili jesteś zalogowany.'
    return 'NIE jesteś zalogowany.'
```

To jest kod, który należy skopiować i wkleić.

Ten kod zwraca różne komunikaty w zależności od tego, czy użytkownik korzystający z przeglądarki jest zalogowany, czy nie.

Poniżej został przedstawiony bieżący kod funkcji `page1`.

```
@app.route('/page1')
def page1():
    return 'To jest strona  1.'
```

To jest kod odpowiedzialny za działanie specyficzne dla strony.

Gdybyśmy zdecydowali się skopiować wyróżniony kod z funkcji `check_status` i wkleić go do funkcji `page1`, ta ostatnia przyjęłaby postać zaprezentowaną poniżej.

```
@app.route('/page1')
def page1():
    if 'logged_in' in session:
        return 'To jest strona  1.'
    return 'NIE jesteś zalogowany.'
```

Sprawdź, czy użytkownik korzystający z przeglądarki jest zalogowany...

...a następnie przeprowadź operacje charakterystyczne dla tej strony.

W przeciwnym razie poinformuj użytkownika, że nie jest zalogowany.

Przedstawiony powyżej kod działa, ale gdybyś musiał powtórzyć tę operację kopiowania i wklejania dla URL-i `/page2` i `/page3` (jak również w przypadku wszystkich innych, które zamierzasz wykorzystywać w swojej aplikacji WWW), wkrótce doprowadziłbyś do *utrzymaniowego koszmaru*. Twoje życie zmieniłoby się w takowy, szczególnie gdy weźmiesz pod uwagę wszystkie działania edycyjne, które musiałbyś wykonać, gdybyś zdecydował się zmienić sposób działania swojego kodu sprawdzającego stan zalogowania (na przykład tak, aby podane przez użytkownika identyfikator i hasło były sprawdzane pod kątem zgodności z danymi przechowywanymi w bazie danych).

Umieść współdzielony kod w osobnej funkcji

Gdy masz kod, którego musisz użyć w wielu różnych miejscach, klasycznym rozwiązaniem problemu z utrzymaniem, wiążącego się nieodłącznie z wszelkimi „szybkimi poprawkami" dokonywanymi metodą kopiowania i wklejania, jest umieszczenie wspólnego kodu w funkcji, która jest następnie wywoływana zgodnie z potrzebami.

Strategia ta umożliwia co prawda rozwiązanie problemu z utrzymaniem (ponieważ współdzielony kod istnieje tylko w jednym miejscu, zamiast być kopiowany i wklejany, gdzie tylko popadnie), sprawdźmy jednak, w jaki sposób utworzenie funkcji sprawdzającej stan zalogowania wpływa na naszą aplikację WWW.

Utworzenie funkcji pomaga, ale...

Utwórzmy nową funkcję o nazwie `check_logged_in`, której wywołanie powoduje zwrócenie wartości `True`, gdy użytkownik korzystający z przeglądarki jest w danej chwili zalogowany, lub `False` — w przeciwnym razie.

Nie jest to zadanie trudne (większość niezbędnego kodu znajduje się już w funkcji `check_status`); poniżej możesz zobaczyć, jak postanowiliśmy napisać tę funkcję.

```
def check_logged_in() -> bool:
    if 'logged_in' in session:
        return True
    return False
```

Zamiast zwracać komunikat, ta funkcja zwraca wartość logiczną w zależności od tego, czy użytkownik korzystający z przeglądarki jest zalogowany, czy też nie.

Mając tę funkcję gotową, skorzystajmy z niej w funkcji `page1` zamiast z kodu, który wcześniej skopiowaliśmy i wkleiliśmy. Nowa wersja funkcji `page1` znajduje się poniżej.

```
@app.route('/page1')
def page1():
    if not check_logged_in():
        return 'NIE jesteś zalogowany.'
    return 'To jest strona  1.
```

Sprawdzamy tutaj, czy NIE jesteśmy zalogowani.

Wywołaj funkcję check_logged_in, aby określić stan zalogowania, a następnie działaj zgodnie z tym stanem.

Ten kod nigdy nie zostanie wykonany, jeśli użytkownik korzystający z przeglądarki jest zalogowany.

Strategia ta jest nieco lepsza niż kopiowanie i wklejanie kodu, ponieważ zmiany sposobu działania procesu logowania możesz teraz dokonać, modyfikując kod funkcji `check_logged_in`. Jednak aby wykorzystać tę funkcję, nadal musisz wprowadzić podobne zmiany w funkcjach `page2` oraz `page3` (jak również w całym kodzie obsługi każdego kolejnego URL-a, jaki tylko dodasz), a zrobisz to, kopiując ten nowy fragment kodu funkcji `page1` i wklejając go w odpowiednich miejscach... W gruncie rzeczy, jeśli porównasz to, co zrobiłeś z funkcją `page1` na tej stronie, z tym, co z nią zrobiłeś na stronie poprzedniej, okaże się, że wiąże się to z koniecznością wykonania niemal tej samej pracy, a co więcej, *w dalszym ciągu* praca ta polega na kopiowaniu i wklejaniu kodu! W dodatku użycie któregokolwiek z tych „rozwiązań” spowoduje, że dodany kod **zaciemni** to, co funkcja `page1` robi naprawdę.

Byłoby miło, gdybyśmy mogli w jakiś sposób sprawdzić, czy użytkownik korzystający z przeglądarki jest zalogowany, *bez* konieczności wprowadzania zmian w kodzie *jakiejkolwiek* z naszych istniejących funkcji (aby niczego nie zaciemniać). Dzięki temu kod znajdujący się w każdej funkcji Twojej aplikacji WWW mógłby nadal być *bezpośrednio* związany wyłącznie z tym, co dana funkcja robi, a kod odpowiedzialny za sprawdzanie stanu zalogowania nie wchodziłby nam w drogę. Gdyby tylko można było to zrobić...

Jak już wiesz od naszych trzech zaprzyjaźnionych programistów — Franka, Józka i Kuby — których rozmowę miałeś okazję niedawno przeczytać, język Python zapewnia mechanizm, który może nam tu pomóc; nosi on nazwę **dekoratora**. Dekorator umożliwia rozszerzenie istniejącej funkcji za pomocą dodatkowego kodu i robi to, pozwalając Ci zmieniać działanie tej funkcji *bez* konieczności modyfikacji jej kodu.

Jeśli przeczytałeś to ostatnie zdanie i zapytałeś: *Co?!?!?!*, nie martw się; zapewne brzmi to dziwnie za pierwszym razem, gdy to słyszysz. W końcu jak możemy zmienić sposób działania funkcji, nie zmieniając jednocześnie jej kodu? Czy jest sens w ogóle czegoś takiego próbować?

Przekonajmy się o tym, poznając dekoratory.

Używasz dekoratorów przez cały czas

Z dekoratorów *korzystasz* już od czasu, gdy zaczęliśmy tworzyć aplikacje WWW przy użyciu frameworka Flask, z którym masz do czynienia od rozdziału 5.

Poniżej znajdziesz najstarszą wersję aplikacji *hello_flask.py*, która powstała we wspomnianym rozdziale. W jej kodzie wyróżniliśmy dekorator o nazwie `@app.route`, z którego możemy korzystać dzięki frameworkowi Flask. Dekorator ten został zastosowany do istniejącej funkcji (tutaj jest to funkcja `hello`), a jego rolą jest taka zmiana działania funkcji, którą poprzedza, aby była ona wywoływana zawsze wtedy, gdy aplikacja przetwarza URL /. Dekoratory łatwo jest dojrzeć w kodzie, ponieważ ich nazwy są poprzedzone symbolem `@`.

Tutaj znajduje się dekorator, który — podobnie jak wszystkie inne dekoratory — jest poprzedzany symbolem @.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Witaj, świecie, tu Flask!'

app.run()
```

Zauważ, że jako użytkownik dekoratora `@app.route` nie masz pojęcia, w jaki sposób czyni on swoją magię. Interesuje Cię wyłącznie, aby robił to, czego się od niego oczekuje, a więc łączył podany URL z funkcją. Wszystkie czary-mary, szczegóły odbywającego się w tle działania, są przed Tobą ukryte.

Gdy postanowisz utworzyć dekorator, będziesz musiał zajrzeć pod maskę i (podobnie jak miało to miejsce wtedy, gdy w poprzednim rozdziale tworzyliśmy menedżer kontekstu) wgłębić się w działanie maszynerii dekoratorów w języku Python. Aby napisać własny dekorator, musisz wiedzieć o czterech ważnych rzeczach i je rozumieć:

1. **Jak utworzyć funkcję.**
2. **Jak przekazać funkcję jako argument funkcji.**
3. **Jak zwrócić funkcję jako wynik działania funkcji.**
4. **Jak przetworzyć dowolną liczbę i dowolny typ argumentów funkcji.**

Z powodzeniem tworzysz swoje funkcje i korzystasz z nich już od rozdziału 4., co oznacza, że nasza lista „czterech rzeczy, które trzeba wiedzieć” tak naprawdę obejmuje tylko trzy nowe punkty. Poświęćmy trochę czasu, aby zapoznać się z punktami od 2. do 4. wraz z tym, jak będziemy samodzielnie tworzyć swój dekorator.

Przekaż funkcję do funkcji

- [ ] Przekaż funkcję do funkcji.
- [ ] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Każdy kolejny zamknięty temat będziemy odhaczać na liście wraz z omawianiem materiału.

Od tego momentu minęło już sporo czasu, ale w rozdziale 2. stwierdziliśmy, że w języku Python *wszystko jest obiektem*. Chociaż może to brzmieć dość paradoksalnie, to „wszystko” obejmuje również funkcje, co oznacza, że także one są obiektami.

Rzecz jasna, gdy wywołujesz funkcję, ona się wykonuje. Jednak podobnie jak wszystko inne w Pythonie, funkcje są obiektami i mają swój identyfikator obiektu. Traktuj je jako „obiekty funkcji”.

Spójrz szybko na krótką sesję IDLE przedstawioną poniżej. Łańcuch znakowy zostaje tu przypisany do zmiennej o nazwie `msg`, po czym jej identyfikator obiektu jest wyświetlany za pomocą wywołania wbudowanej funkcji `id`. Następnie zostaje zdefiniowana niewielka funkcja o nazwie `hello`. Również ona jest przekazywana do funkcji `id`, która wyświetla identyfikator obiektu funkcji. Następnie zostaje dwukrotnie użyta wbudowana funkcja `type` w celu potwierdzenia, że zmienna `msg` jest łańcuchem znakowym, a `hello` — funkcją. Na samym końcu zostaje wywołana funkcja `hello`, która wyświetla wartość zmiennej `msg` na ekranie.

Wbudowana funkcja id wyświetla unikatowy identyfikator obiektu dla dowolnego obiektu, który zostaje jej przekazany.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> msg = "Witaj, tu Python. Rusz głową!"
>>> id(msg)
2861340453856
>>> def hello():
        print(msg)


>>> id(hello)
2861340453744
>>> type(msg)
<class 'str'>
>>> type(hello)
<class 'function'>
>>> hello()
Witaj, tu Python. Rusz głową!
>>>
```

Ln: 67 Col: 4

Wbudowana funkcja type wyświetla typ obiektu.

Przed poproszeniem Cię o przyjrzenie się powyższej sesji IDLE dość przebiegle nie zwróciliśmy Twojej uwagi na pewną rzecz, ale... czy zauważyłeś, jak przekazaliśmy funkcję `hello` do wbudowanych funkcji `id` oraz `type`? Nie wywołaliśmy jej, lecz przekazaliśmy jej *nazwę* do każdej z tych funkcji jako argument. W ten sposób przekazaliśmy funkcję do funkcji.

Funkcje mogą przyjmować funkcję jako argument

Te przedstawione wyżej wywołania funkcji `id` i `type` pokazują, że niektóre wbudowane funkcje języka Python w roli swojego argumentu akceptują funkcję (lub raczej — bardziej precyzyjnie — *obiekt funkcji*). Co funkcja robi z takim argumentem, to już jej sprawa. Ani funkcja `id`, ani `type` nie wywołuje przekazanej jej funkcji, chociaż w zasadzie mogłaby to robić. Zobaczmy, jak działa ten mechanizm.

Wywoływanie przekazanej funkcji

- [ ] Przekaż funkcję do funkcji.
- [ ] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Gdy obiekt funkcji zostaje przekazany jako argument do innej funkcji, ta ostatnia może *wywołać* przekazany jej obiekt funkcji.

Poniżej znajdziesz kod niewielkiej funkcji (o nazwie `apply`), która przyjmuje dwa argumenty: obiekt funkcji i wartość. Funkcja `apply` wywołuje otrzymany obiekt funkcji i przekazuje mu otrzymaną wartość jako argument, zwracając wyniki wywołania funkcji z wartością do kodu, który ją wywołuje.

Funkcja apply przyjmuje obiekt funkcji jako argument. Adnotacja object pomaga tu potwierdzić nasze intencje (a użycie nazwy „func" w przypadku argumentu stanowi ogólnie przyjętą konwencję).

func.py - C:\Users\lsuma\Desktop\Python\R10\func.py (3.6.0)

File Edit Format Run Options Window Help

```
def apply(func: object, value: object) -> object:
    return func(value)
```

Ln: 6 Col: 0

Dowolna wartość (każdego typu) może być przekazana jako drugi argument funkcji. Również tutaj adnotacja wskazuje, co jest dozwolone w roli typu argumentu; jest to dowolny obiekt.

Funkcja (przekazana jako argument) zostaje wywołana z jedynym argumentem, którym jest value. Wynik jej wywołania zostaje zwrócony przez funkcję apply.

Zwróć uwagę na to, jak adnotacje funkcji `apply` wskazują, że przyjmuje ona dowolny obiekt funkcji oraz dowolną wartość, a następnie zwraca cokolwiek (co jest niezwykle *ogólne*). Szybki test funkcji `apply` w wierszu poleceń potwierdza, że działa ona zgodnie z oczekiwaniami, tak jak zostało to przedstawione poniżej.

Funkcja apply zostaje użyta do uruchomienia kilku funkcji wbudowanych z pewnymi wartościami (i działa przy tym zgodnie z oczekiwaniami).

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> apply(print, 42)
42
>>> apply(id, 42)
1580706176
>>> apply(type, 42)
<class 'int'>
>>> apply(len, 'Marvin')
6
>>> apply(type, apply)
<class 'function'>
>>>
```

Ln: 83 Col: 4

Funkcja apply przyjmuje dowolny obiekt jako argument value. W tym przykładzie przyjmuje samą siebie jako argument value i potwierdza, że jest funkcją.

W każdym z tych przykładów pierwszy argument funkcji apply jest przypisywany do argumentu func.

Jeśli czytasz tę stronę i zastanawiasz się, kiedy w ogóle miałoby Ci się przydać coś takiego, nie irytuj się niepotrzebnie, ponieważ dojdziemy do tego, gdy zaczniemy pisać swój dekorator. Na razie skup się na tym, aby zrozumieć, że do funkcji można przekazać obiekt funkcji, który może być później wywołany.

Funkcje mogą być zagnieżdżone wewnątrz funkcji

- [x] Przekaż funkcję do funkcji.
- [ ] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Zwykle gdy tworzysz funkcję, korzystasz z jakiegoś istniejącego kodu i nadając mu nazwę oraz wykorzystując go w roli zestawu funkcji, sprawiasz, że można go ponownie używać. Z tym przypadkiem mamy do czynienia najczęściej. Jednak, choć może to być dla Ciebie pewną niespodzianką, zestaw funkcji w Pythonie może zawierać *dowolny* kod, również taki, który stanowi definicję zupełnie innej funkcji (określanej jako funkcja *zagnieżdżona* lub *wewnętrzna*). Jeszcze bardziej zaskakujące jest to, że zagnieżdżona funkcja może być *zwracana* przez zewnętrzną, obejmującą ją funkcję. Skutkiem tego zwracany jest *obiekt funkcji*. Przyjrzymy się kilku przykładom prezentującym ten inny, mniej spotykany przypadek zastosowania funkcji.

Pierwszym przykładem jest zaprezentowany poniżej fragment kodu, który pokazuje funkcję (o nazwie `inner`) zagnieżdżoną w innej funkcji (o nazwie `outer`). Pierwszej z wymienionych funkcji nie da się wywołać z żadnego innego miejsca niż wnętrze zestawu drugiej, ponieważ funkcja `inner` ma charakter lokalny w zasięgu funkcji `outer`.

Funkcja inner jest zdefiniowana wewnątrz zestawu funkcji obejmującej.

```
def outer():
    def inner():
        print('To jest funkcja inner.')

    print('To jest funkcja outer, wywołująca funkcję inner.')
    inner()
```

Funkcja inner jest wywoływana z wnętrza funkcji outer.

Gdy funkcja `outer` zostaje wywołana, wykonuje cały kod znajdujący się w jej zestawie, co oznacza, że zostaje zdefiniowana funkcja `inner`, wywołana wbudowana funkcja `print`, a następnie wywołana funkcja `inner` (która wywołuje z kolei w swoim zestawie wbudowaną funkcję `print`). Poniżej możesz zobaczyć, jaki jest efekt tego działania na ekranie.

```
To jest funkcja outer, wywołująca funkcję inner.
To jest funkcja inner.
```

Komunikaty wyświetlane są na ekranie w następującej kolejności: najpierw outer, następnie inner.

Kiedy miałbyś z tego w ogóle korzystać?

Gdy patrzysz na ten prosty przykład, może Ci być trudno wyobrazić sobie sytuację, w której tworzenie funkcji wewnątrz innej funkcji miałoby się okazać w czymś pomocne. Jednak gdy funkcja jest skomplikowana i zawiera wiele wierszy kodu, wydzielenie jego części w postaci funkcji zagnieżdżonej ma zwykle sens (i może sprawić, że kod funkcji obejmującej będzie się znacznie łatwiej czytać).

Częstszym zastosowaniem tej techniki jest zwracanie zagnieżdżonej funkcji przez obejmującą jako jej wartości, co realizuje się za pomocą instrukcji `return`. To właśnie ten mechanizm umożliwia Ci tworzenie dekoratorów.

Zobaczmy zatem, co się dzieje, gdy przez funkcję zwracana jest funkcja.

Zwróć funkcję z funkcji

- [x] Przekaż funkcję do funkcji.
- [ ] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Drugi prezentowany przez nas przykład bardzo przypomina pierwszy, tym razem jednak funkcja `outer` zamiast wywoływać funkcję `inner`, zwraca ją. Przyjrzyj się przedstawionemu poniżej kodowi.

Funkcja inner nadal jest zdefiniowana wewnątrz zestawu funkcji outer.

```
def outer():
    def inner():
        print('To jest funkcja inner.')

    print('To jest funkcja outer, wywołująca funkcję inner.')
    return inner()
```

Instrukcja return nie wywołuje funkcji inner, lecz zwraca obiekt funkcji inner do wywołującego kodu.

Przekonajmy się, co robi ta nowa wersja funkcji `outer` — powróćmy do okna powłoki środowiska IDLE i weźmy ją w obroty.

Zauważ, że w przedstawionym poniżej przykładzie przypisujemy wynik wywołania funkcji `outer` do zmiennej o nazwie `i`. Następnie używamy tej zmiennej w taki sposób, jakby był to obiekt funkcji — najpierw wywołujemy wbudowaną funkcję `type` z argumentem `i`, a następnie wywołujemy `i` tak, jak zrobilibyśmy to z każdą inną funkcją (tzn. przez podanie nazwy, po której występuje nawias okrągły). Gdy robimy to ostatnie, wykonywana jest funkcja `inner`. W rezultacie `i` stanowi teraz synonim funkcji `inner` utworzonej wewnątrz funkcji `outer`.

Funkcja outer jest wywoływana.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> i = outer()
To jest funkcja outer, zwracająca funkcję inner.
>>> type(i)
<class 'function'>
>>> i()
To jest funkcja inner.
>>>
>>>
```

Ln: 126 Col: 4

Wynik wywołania funkcji outer zostaje przypisany do zmiennej i.

Przekonaliśmy się, że i jest w rzeczywistości funkcją.

Wywołujemy zmienną i jak funkcję i — voilà! — wykonywany jest kod funkcji inner.

Jak dotąd świetnie. Teraz wiesz już, jak *zwraca* się funkcję z funkcji, wiesz też, jak *przesyła* się funkcję do funkcji. Jesteś już niemal gotów połączyć te informacje i ruszyć z tworzeniem dekoratora. Musisz tylko jeszcze zrozumieć jedną kwestię, a mianowicie tworzenie funkcji, która radzi sobie z dowolną liczbą i typem argumentów. Przyjrzyjmy się teraz temu zagadnieniu.

Przyjmowanie listy argumentów

- [x] Przekaż funkcję do funkcji.
- [x] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Jesteś niemal u celu. Jeszcze jeden temat do omówienia i będziesz gotów do utworzenia dekoratora.

Wyobraź sobie, że masz utworzyć funkcję (będziemy ją w tym przykładzie nazywać `myfunc`), którą można wywoływać z dowolną liczbą argumentów. Funkcję tę możesz na przykład wywołać w następujący sposób:

```
myfunc(10)
```

Jeden argument

albo w sposób pokazany poniżej:

```
myfunc()
```

Brak argumentów

lub też tak, jak widać tutaj:

```
myfunc(10, 20, 30, 40, 50, 60, 70)
```

Wiele argumentów (którymi w tym przykładzie są wyłącznie liczby, ale równie dobrze argumentami mogłoby być cokolwiek innego: łańcuchy znakowe, wartości logiczne lub też listy)

W rzeczywistości funkcję `myfunc` powinno się dać wywołać z *dowolną* liczbą argumentów, przy czym nie będziesz wiedział z góry, ile tych argumentów zostanie podanych.

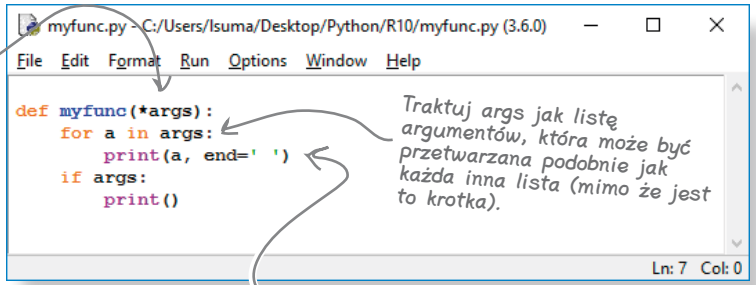
Jako że w języku Python nie da się zdefiniować trzech różnych wersji funkcji `myfunc`, które obsłużyłyby trzy przedstawione powyżej wywołania, pojawia się pytanie: *jak zapewnić funkcji możliwość przyjmowania dowolnej liczby argumentów?*

Użyj symbolu *, aby funkcja mogła przyjmować dowolną listę argumentów

Język Python zapewnia specjalny zapis umożliwiający określenie, że funkcja może przyjmować dowolną liczbę argumentów (przy czym jako „dowolną liczbę” rozumie się tu „zero lub więcej”). Zapis ten wykorzystuje znak * do reprezentowania *dowolnej liczby* i łączy się go z nazwą argumentu (którą zgodnie z konwencją jest `args`), aby określić, że funkcja może przyjmować dowolną listę argumentów (mimo że `*args` z technicznego punktu widzenia jest krotką).

Przyjmij, że symbol * oznacza rozwinięcie do listy wartości.

Poniżej znajdziesz wersję funkcji `myfunc`, w której skorzystano z tego zapisu, aby mogła przyjmować dowolną liczbę argumentów przy wywołaniu. Jeśli zostaną podane jakiekolwiek argumenty, funkcja ta wyświetli ich wartości na ekranie.



Zapis *args oznacza „zero lub więcej argumentów”.

Traktuj args jak listę argumentów, która może być przetwarzana podobnie jak każda inna lista (mimo że jest to krotka).

Wyświetla listę wartości argumentów w jednym wierszu.

Przetwarzanie listy argumentów

- [x] Przekaż funkcję do funkcji.
- [x] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Gdy istnieje już funkcja `myfunc`, sprawdźmy, czy jest ona w stanie poradzić sobie z przykładowymi wywołaniami przedstawionymi na poprzedniej stronie, jak również poniżej.

```
myfunc(10)
myfunc()
myfunc(10, 20, 30, 40, 50, 60, 70)
```

Poniżej znajdziesz zapis kolejnej sesji IDLE, która stanowi potwierdzenie, że funkcja `myfunc` radzi sobie świetnie. Niezależnie od liczby dostarczonych argumentów (również wtedy, gdy nie ma *żadnego*) funkcja odpowiednio je przetwarza, tak jak zostało to pokazane poniżej.

Nieważne, ile argumentów się poda, funkcja myfunc robi to, co trzeba (tzn. przetwarza swoje argumenty niezależnie od ich liczby).

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> myfunc(10)
10
>>> myfunc()
>>> myfunc(10, 20, 30, 40, 50, 60, 70)
10 20 30 40 50 60 70
>>> myfunc(1, 'dwa', 3, 'cztery', 5, 'sześć', 7)
1 dwa 3 cztery 5 sześć 7
>>>
```

Ln: 138 Col: 4

Wywołana bez argumentów funkcja myfunc nie wykonuje żadnej operacji.

Możesz nawet podawać wartości różnych typów w jednym wywołaniu, a funkcja myfunc nadal robi to, co należy.

Symbol * działa również przy wywoływaniu funkcji

Gdy jako argument funkcji `myfunc` podajesz listę (zawierającą potencjalnie wiele wartości), traktowana jest ona jako jeden element (ponieważ w istocie *jest* jedną listą). Aby poinformować interpreter, że ma **rozwinąć** listę w taki sposób, aby jej elementy składowe były traktowane jako *oddzielne* argumenty, przed nazwą listy umieść znak *, gdy wywołujesz funkcję.

Kolejna, przedstawiona poniżej sesja IDLE pokazuje różnicę wynikającą z zastosowania symbolu * przy wywołaniu funkcji.

Lista jest przetwarzana jako jeden argument funkcji.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> values = [1, 2, 3, 5, 7, 11]
>>>
>>> myfunc(values)
[1, 2, 3, 5, 7, 11]
>>> myfunc(*values)
1 2 3 5 7 11
>>>
>>>
```

Ln: 147 Col: 4

Lista składa się z sześciu liczb całkowitych.

Gdy listę poprzedza znak *, jest ona rozwijana do postaci sześciu oddzielnych argumentów.

Przyjmowanie słowników argumentów

- [x] Przekaż funkcję do funkcji.
- [x] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Jeśli chodzi o przesyłanie wartości do funkcji, możliwe jest również podawanie nazw argumentów wraz ze związanymi z nimi wartościami i pozostawienie interpreterowi odpowiedniego dopasowania ich w wywołaniu funkcji.

Ze sposobem tym spotkałeś się po raz pierwszy w rozdziale 4., gdy użyliśmy go w wywołaniu funkcji `search4letters`, która — jak być może pamiętasz — oczekuje podania dwóch wartości argumentów, jednej dla argumentu `phrase`, a drugiej dla argumentu `letters`. Gdy korzysta się ze słów kluczowych argumentów, kolejność argumentów przestaje mieć znaczenie.

To jest jeden ze sposobów wywołania tej funkcji.

```
search4letters(letters='tym', phrase='galaktyka')
```

A to inny sposób jej wywołania.

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Podobnie jak ma to miejsce w przypadku zastosowania list, można również sprawić, aby funkcja przyjmowała dowolną liczbę słów kluczowych argumentów, czyli kluczy wraz z powiązanymi z nimi wartościami (dokładnie tak, jak w przedstawionym powyżej przykładzie z argumentami `phrase` i `letters`).

Użyj znaków **, aby funkcja przyjmowała dowolne argumenty ze słowami kluczowymi

Oprócz symbolu * język Python umożliwia stosowanie zapisu **, który powoduje rozwinięcie argumentu do postaci kolekcji argumentów ze słowami kluczowymi. Podczas gdy w przypadku * zgodnie z konwencją korzysta się z nazwy zmiennej `args`, przy stosowaniu ** używa się nazwy `kwargs`, stanowiącej skrót od angielskich słów *keyword arguments*, czyli „argumenty ze słowami kluczowymi". (Zauważ, że możesz w tym kontekście stosować inne nazwy niż `args` i `kwargs`, robi to jednak bardzo niewielu programistów Pythona).

Przyjmij, że zapis ** oznacza rozwinięcie do słownika zawierającego klucze i wartości.

Przyjrzyjmy się innej funkcji, o nazwie `myfunc2`, która przyjmuje dowolną liczbę argumentów ze słowami kluczowymi.

myfunc.py - C:/Users/lsuma/Desktop/Python/R10/myfunc.py (3.6.0)

File Edit Format Run Options Window Help

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()

def myfunc2(**kwargs):
    for k, v in kwargs.items():
        print(k, v, sep='->', end=' ')
    if kwargs:
        print()
```

Ln: 13 Col: 0

Zapis ** informuje funkcję, że ma się spodziewać argumentów ze słowami kluczowymi.

Weź każdą parę klucz-wartość ze słownika i wyświetl ją na ekranie.

Wewnątrz funkcji argument kwargs zachowuje się dokładnie tak, jak każdy inny słownik.

Przetwarzanie słownika argumentów

- [x] Przekaż funkcję do funkcji.
- [x] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Kod należący do zestawu funkcji `myfunc2` bierze słownik argumentów i przetwarza je, wyświetlając wszystkie pary klucz-wartość w pojedynczym wierszu na ekranie.

Poniżej możesz zobaczyć zapis kolejnej sesji IDLE, która prezentuje działanie tej funkcji w praktyce. Niezależnie od tego, ile poda się par klucz-wartość (w tym również, gdy nie poda się żadnej), funkcja `myfunc2` robi to, co do niej należy.

Tu podano dwa argumenty ze słowami kluczowymi.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>>
>>> myfunc2(a=10, b=20)
a->10 b->20
>>> myfunc2()
>>> myfunc2(a=10, b=20, c=30, d=40, e=50, f=60)
a->10 b->20 c->30 d->40 e->50 f->60
>>>
>>>
Ln: 157  Col: 4
```

Niepodanie żadnych argumentów nie stanowi problemu.

Funkcji myfunc2 możesz podać dowolną liczbę argumentów ze słowami kluczowymi i zawsze zrobi to, co trzeba.

Zapis ** działa również przy wywoływaniu funkcji

Zastanawiałeś się pewnie, czy tutaj też tak będzie, prawda? Podobnie jak w przypadku argumentu `*args`, również gdy korzystasz z argumentu `**kwargs`, możesz używać zapisu ** podczas wywoływania funkcji `myfunc2`. Zamiast demonstrować, jak działa to w tej konkretnej funkcji, przypomnimy Ci wcześniejsze zastosowanie tego sposobu, z którym miałeś już do czynienia w tej książce. W rozdziale 7., gdy poznawałeś DB-API języka Python, definiowałeś słownik właściwości połączenia, tak jak pokazano poniżej.

To jest słownik zawierający pary klucz-wartość.

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }
```

Ze słownika tego korzystałeś w przedstawiony poniżej sposób, gdy zachodziła potrzeba nawiązania połączenia ze swoim oczekującym serwerem bazy danych MySQL (lub MariaDB). Czy zauważyłeś, w jaki sposób określony tu został argument `dbconfig`?

Czy wygląda to znajomo?

```
conn = mysql.connector.connect(**dbconfig)
```

Poprzedzając argument `dbconfig` znakami **, poinformowaliśmy interpreter, że ma traktować pojedynczy słownik jak kolekcję kluczy i powiązanych z nimi wartości. W wyniku tego wywołanie funkcji `connect` odbyło się tak, jakbyś podał w nim cztery oddzielne argumenty ze słowami kluczowymi, co pokazano poniżej.

```
conn = mysql.connector.connect('host'='127.0.0.1', 'user'='vsearch',
                               'password'='vsearchpasswd', 'database'='vsearchlogDB')
```

Przyjmowanie dowolnej liczby i typu argumentów funkcji

- [x] Przekaż funkcję do funkcji.
- [x] Zwróć funkcję z funkcji.
- [ ] Przetwórz dowolną liczbę i typ argumentów.

Bardzo miłe ze strony Pythona jest zarówno to, że umożliwia przyjmowanie przez tworzone przez Ciebie funkcje listy argumentów (za pomocą symbolu *), jak i to, że pozwala na podawanie dowolnej liczby argumentów ze słowami kluczowymi (przy użyciu zapisu **). Co jeszcze lepsze, możesz łączyć te dwa sposoby, dzięki czemu jesteś w stanie utworzyć funkcję, która przyjmuje dowolną liczbę i typ argumentów.

Poniżej znajdziesz trzecią wersję funkcji `myfunc` (która nosi szokująco wymyślną nazwę `myfunc3`). Przyjmuje ona dowolną listę argumentów, dowolną liczbę argumentów ze słowami kluczowymi lub połączenie obydwu tych rzeczy.

Oryginalna funkcja myfunc umożliwia podanie dowolnej listy argumentów.

Funkcja myfunc2 umożliwia podanie dowolnej liczby par klucz-wartość.

Funkcja myfunc3 umożliwia podanie dowolnych danych wejściowych, którymi mogą być zarówno lista argumentów, wiele par klucz-wartość, jak i kombinacja tych dwóch rzeczy.

W wierszu def tej funkcji znajdują się zarówno argument *args, jak i **kwargs.

myfunc.py - C:/Users/lsuma/Desktop/Python...

File Edit Format Run Options Window Help

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()

def myfunc2(**kwargs):
    for k, v in kwargs.items():
        print(k, v, sep='->', end=' ')
    if kwargs:
        print()

def myfunc3(*args, **kwargs):
    if args:
        for a in args:
            print(a, end=' ')
        print()
    if kwargs:
        for k, v in kwargs.items():
            print(k, v, sep='->', end=' ')
        print()
```

Ln: 23 Col: 0

Przedstawiona poniżej krótka sesja IDLE pokazuje sposób działania funkcji `myfunc3`.

Działa bez argumentów.

Działa z listą.

Działa z argumentami ze słowami kluczowymi.

Działa z kombinacją listy i argumentów ze słowami kluczowymi.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> myfunc3()
>>> myfunc3(1, 2, 3)
1 2 3
>>> myfunc3(a=10, b=20, c=30)
a->10 b->20 c->30
>>> myfunc3(1, 2, 3, a=10, b=20, c=30)
1 2 3
a->10 b->20 c->30
>>>
```

Ln: 169 Col: 4

Przepis na opracowanie dekoratora funkcji

- ✔ Przekaż funkcję do funkcji.
- ✔ Zwróć funkcję z funkcji.
- ✔ Przetwórz dowolną liczbę i typ argumentów.

Jesteśmy gotowi do rozpoczęcia pracy nad własnym dekoratorem.

Skoro odhaczyłeś już trzy punkty na naszej liście widocznej z prawej strony, masz już wystarczającą wiedzę na temat możliwości języka Python, które pozwolą Ci utworzyć dekorator. Musisz już tylko dowiedzieć się, jak z nich skorzystać i w jaki sposób je połączyć, aby opracować potrzebny Ci dekorator.

Podobnie jak miało to miejsce w przypadku tworzenia własnego menedżera kontekstu (czym zajmowaliśmy się w poprzednim rozdziale), procesem tworzenia dekoratora rządzi pewien zestaw reguł czy też *przepis*. Przypomnij sobie, że dekorator umożliwia modyfikację istniejącej funkcji za pomocą dodatkowego kodu, jednak bez konieczności zmiany kodu tej funkcji (co, jak sami przyznajemy, nadal brzmi dość dziwacznie).

Aby opracować dekorator funkcji, musisz wiedzieć następujące rzeczy:

1. **Dekorator jest funkcją.**
   W rzeczywistości z punktu widzenia interpretera dekorator jest *po prostu kolejną funkcją*, aczkolwiek taką, która manipuluje inną, istniejącą już funkcją. Tę ostatnią będziemy od tej pory określać mianem *funkcji dekorowanej*. Dzięki lekturze wcześniejszej części tej książki wiesz już, że tworzenie funkcji to nic trudnego; wystarczy tu skorzystać ze słowa kluczowego `def` języka Python.

2. **Dekorator przyjmuje dekorowaną funkcję jako argument.**
   Dekorator musi przyjmować dekorowaną funkcję jako swój argument. Aby ją do niego przesłać, powinieneś po prostu przekazać do dekoratora funkcję dekorowaną jako *obiekt funkcji*. Po lekturze ostatnich 10 stron tej książki wiesz już, że to też jest proste; do obiektu funkcji można się odwołać, używając samej jej nazwy *bez* nawiasów (ponieważ nawiasy informują interpreter, że funkcja ma być wywołana).

3. **Dekorator zwraca nową funkcję.**
   Dekorator zwraca nową funkcję jako swoją wartość zwracaną. Odbywa się to podobnie, jak było w przypadku funkcji `outer`, która zwracała funkcję `inner` (kilka stron wcześniej). Dekorator działa bardzo podobnie, z tą różnicą, że zwracana funkcja *musi* wywoływać funkcję dekorowaną. Zapewnienie tego jest — *wybacz, że to piszemy* — rzeczą łatwą, lecz wiąże się z pewną drobną komplikacją, której poświęcimy cały następny punkt tej listy.

4. **Dekorator zachowuje sygnaturę funkcji dekorowanej.**
   Dekorator musi zapewniać, że zwracana przez niego funkcja przyjmuje taką samą liczbę i typy argumentów, jakich spodziewa się funkcja dekorowana. Liczba i typy argumentów każdej funkcji znane są jako jej **sygnatura** (ponieważ wiersz `def` każdej funkcji ma charakter unikatowy).

Nadszedł czas na wzięcie do ręki ołówka i skorzystanie z zaprezentowanych powyżej informacji w celu utworzenia pierwszego dekoratora.

Przypomnienie: mamy ograniczyć dostęp do określonych URL-i

Pracowaliśmy nad kodem aplikacji WWW o nazwie *simple_webapp.py* i okazało się, że potrzebujemy dekoratora do sprawdzania, czy użytkownik korzystający z przeglądarki internetowej jest zalogowany, czy też nie. Jeśli jest, powinna mu zostać pokazana strona o ograniczonym dostępie. Jeśli nie jest, aplikacja WWW powinna powiadomić użytkownika, że musi się zalogować, zanim uzyska dostęp do tego rodzaju stron. Aby obsłużyć tę logikę, opracujemy dekorator. Przypomnij sobie przedstawioną poniżej funkcję `check_status`, która demonstrowała logikę, którą ma naśladować nasz dekorator.

Chcemy uniknąć kopiowania i wklejania tego kodu.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'W tej chwili jesteś zalogowany.'
    return 'NIE jesteś zalogowany.'
```

Pamiętaj, że ten kod zwraca różne komunikaty w zależności od tego, czy użytkownik przeglądarki jest zalogowany, czy też nie.

Tworzenie dekoratora funkcji

Aby spełnić wymóg stawiany przez 1. punkt naszej listy, musisz utworzyć nową funkcję. Pamiętaj, że:

1. **Dekorator jest funkcją.**
W rzeczywistości z punktu widzenia interpretera dekorator jest *po prostu kolejną funkcją*, aczkolwiek taką, która manipuluje inną, istniejącą już funkcją. Tę ostatnią będziemy od tej pory określać mianem *funkcji dekorowanej*. Dzięki lekturze wcześniejszej części tej książki wiesz już, że tworzenie funkcji to nic trudnego; wystarczy tu skorzystać ze słowa kluczowego `def` języka Python.

Spełnienie wymogu określonego w punkcie 2. polega na sprawieniu, aby dekorator przyjmował obiekt funkcji jako swój argument. Ponownie, pamiętaj, że:

2. **Dekorator przyjmuje dekorowaną funkcję jako argument.**
Dekorator musi przyjmować dekorowaną funkcję jako swój argument. Aby ją do niego przesłać, powinieneś po prostu przekazać do dekoratora funkcję dekorowaną jako *obiekt funkcji*. Po lekturze ostatnich 10 stron tej książki wiesz już, że to też jest proste; do obiektu funkcji można się odwołać, używając samej jej nazwy *bez* nawiasów (ponieważ nawiasy informują interpreter, że funkcja ma być wywołana).

Zaostrz ołówek

Umieśćmy nasz dekorator w osobnym, jego własnym module (aby dało się go łatwiej ponownie wykorzystywać). Zacznij od utworzenia nowego pliku o nazwie *checker.py* za pomocą swojego edytora kodu.

W pliku tym powinieneś opracować dekorator o nazwie `check_logged_in`. W miejscu zapewnionym poniżej wprowadź kod stanowiący wiersz `def` tego dekoratora. Wskazówka: jako nazwy swojego argumentu odpowiedzialnego za przekazanie obiektu funkcji użyj ciągu `func`.

Umieść wiersz def dekoratora w tym miejscu.

.....

Nie istnieją głupie pytania

P: Czy to, gdzie w moim systemie utworzę plik checker.py, ma jakieś znaczenie?

O: Tak. Planujemy importować plik *checker.py* w kodzie aplikacji WWW, która będzie go potrzebować, dlatego powinieneś zapewnić, żeby interpreter był w stanie odszukać ten plik, gdy w Twoim kodzie pojawi się instrukcja `import checker`. Na razie umieść plik *checker.py* w tym samym folderze, w którym znajduje się plik *simple_webapp.py*.

Zaostrz ołówek
Rozwiązanie

Postanowiliśmy umieścić nasz dekorator w osobnym module (aby dało się go łatwiej ponownie wykorzystywać).

Zacząłeś od utworzenia nowego pliku o nazwie *checker.py* za pomocą swojego edytora kodu.

Twój nowy dekorator (znajdujący się w pliku *checker.py*) nosi nazwę `check_logged_in`. W miejscu zapewnionym poniżej miałeś wprowadzić kod stanowiący wiersz `def` tego dekoratora.

```
def check_logged_in(func):
```

Dekorator check_logged_in przyjmuje pojedynczy argument, którym jest obiekt dekorowanej funkcji.

To niemal zbyt proste, czyż nie?

Pamiętaj, że dekorator to *po prostu kolejna funkcja*, która przyjmuje obiekt funkcji jako swój argument (noszący w przypadku przedstawionego powyżej wiersza `def` nazwę `func`).

Przejdźmy do następnego punktu naszego „przepisu tworzenia dekoratora", który jest nieco bardziej skomplikowany (ale tylko nieco). Przypomnij sobie, co powinien robić Twój dekorator:

3. **Dekorator zwraca nową funkcję.**
   Dekorator zwraca nową funkcję jako swoją wartość zwracaną. Odbywa się to podobnie, jak było w przypadku funkcji `outer`, która zwracała funkcję `inner` (kilka stron wcześniej). Dekorator działa bardzo podobnie, z tą różnicą, że zwracana funkcja *musi* wywoływać funkcję dekorowaną.

We wcześniejszej części tego rozdziału miałeś do czynienia z funkcją `outer`, której wywołanie powodowało zwrócenie funkcji `inner`. Poniżej jeszcze raz zaprezentowaliśmy kod tej funkcji.

```
def outer():
    def inner():
        print('To jest funkcja inner.')

    print('To jest funkcja outer, wywołująca funkcję inner.')
    return inner
```

Funkcja inner jest zagnieżdżona wewnątrz funkcji outer.

Cały ten kod znajduje się wewnątrz zestawu funkcji outer.

Obiekt funkcji inner jest zwracany jako wynik wywołania funkcji outer. Zauważ brak nawiasów po słowie inner wynikający z tego, że jest tu zwracany obiekt funkcji, a NIE wywoływana funkcja inner.

Zaostrz ołówek

Gdy napisałeś już wiersz `def` swojego dekoratora, dodaj do jego zestawu trochę kodu. Powinieneś zrobić tu cztery rzeczy.

1. Zdefiniuj zagnieżdżoną funkcję o nazwie `wrapper`, która jest zwracana przez funkcję `check_logged_in`. (Mógłbyś tu użyć dowolnej nazwy, ale — jak przekonasz się już wkrótce — nazwa `wrapper` jest całkiem dobrym wyborem).

2. Do funkcji `wrapper` dodaj trochę kodu pochodzącego z Twojej istniejącej funkcji `check_status`. Chodzi tu o kod implementujący jedno z dwóch działań uzależnionych od tego, czy użytkownik przeglądarki jest zalogowany, czy też nie. Aby zaoszczędzić Ci przerzucania kartek, poniżej raz jeszcze zamieściliśmy kod funkcji `check_status` (istotne części tego kodu zostały wyróżnione).

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'W tej chwili jesteś zalogowany.'
    return 'NIE jesteś zalogowany.'
```

3. Zgodnie z 3. punktem naszego przepisu tworzenia dekoratora powinieneś sprawić, aby zagnieżdżona funkcja wywoływała funkcję dekorowaną (zamiast zwracać komunikat o treści „W tej chwili jesteś zalogowany", jak ma to miejsce powyżej).

4. Gdy zagnieżdżona funkcja będzie już gotowa, powinieneś zapewnić, aby jej obiekt był zwracany przez funkcję `check_logged_in`.

Dodaj niezbędny kod do zestawu funkcji `check_logged_in` we wskazanych poniżej miejscach.

```
def check_logged_in(func):
```

1. Zdefiniuj zagnieżdżoną funkcję.

.....

.....

.....

.....

.....

2. i 3. Dodaj kod, który ma być wykonywany przez zagnieżdżoną funkcję.

4. Nie zapomnij o zwróceniu zagnieżdżonej funkcji.

Zaostrz ołówek
Rozwiązanie

Po napisaniu wiersza `def` swojego dekoratora miałeś dodać do jego zestawu trochę kodu. Powinieneś był zrobić tu cztery rzeczy.

1. Miałeś zdefiniować zagnieżdżoną funkcję o nazwie `wrapper`, która jest zwracana przez funkcję `check_logged_in`.
2. Do funkcji `wrapper` miałeś dodać trochę kodu pochodzącego z Twojej istniejącej funkcji `check_status`. Chodziło tu o kod implementujący jedno z dwóch działań uzależnionych od tego, czy użytkownik przeglądarki jest zalogowany, czy też nie.
3. Zgodnie z 3. punktem naszego przepisu tworzenia dekoratora powinieneś był sprawić, aby zagnieżdżona funkcja wywoływała funkcję dekorowaną (zamiast zwracać komunikat o treści „W tej chwili jesteś zalogowany", jak ma to miejsce powyżej).
4. Gdy zagnieżdżona funkcja była już gotowa, miałeś zapewnić, aby jej obiekt funkcji był zwracany przez funkcję `check_logged_in`.

Miałeś dodać niezbędny kod do zestawu funkcji `check_logged_in` we wskazanych poniżej miejscach.

```
def check_logged_in(func):
    def wrapper():
        if 'logged_in' in session:
            return func()
        return 'NIE jesteś zalogowany.'
    return wrapper
```

Zagnieżdżony wiersz def zaczyna definicję funkcji wrapper.

Jeśli użytkownik przeglądarki jest zalogowany...

...wywołaj dekorowaną funkcję.

Jeśli użytkownik przeglądarki nie jest zalogowany, zwróć odpowiedni komunikat.

Czy pamiętałeś o zwróceniu zagnieżdżonej funkcji?

Czy rozumiesz już, dlaczego zagnieżdżona funkcja nosi nazwę wrapper?

Jeśli poświęcisz chwilę na przestudiowanie kodu dekoratora (w takiej postaci, jaką ma on obecnie), zauważysz, że zagnieżdżona funkcja nie tylko wywołuje funkcję dekorowaną (przechowywaną w zmiennej `func`), lecz również zmienia jej działanie przez *opakowanie* (ang. *wrapping*) jej wywołania dodatkowym kodem. W tym przypadku ten dodatkowy kod sprawdza, czy klucz `logged_in` znajduje się w słowniku `session` należącym do aplikacji WWW. Co ważne, gdy użytkownik przeglądarki *nie* jest zalogowany, dekorowana funkcja *nigdy* nie zostanie wywołana przez funkcję `wrapper`.

Ostatni krok: obsługa argumentów

Dotarliśmy niemal do celu — „bebechy” kodu naszego dekoratora są już na swoim miejscu. Pozostało już tylko zapewnić, aby dekorator prawidłowo obsługiwał argumenty dekorowanej funkcji bez względu na to, jakie miałyby one być. Przypomnij sobie 4. punkt przepisu:

4. **Dekorator zachowuje sygnaturę funkcji dekorowanej.**
Dekorator musi zapewniać, że zwracana przez niego funkcja przyjmuje taką samą liczbę i typy argumentów, jakich spodziewa się funkcja dekorowana.

Gdy dekorator zostaje zastosowany do istniejącej funkcji, wszystkie jej wywołania są **zastępowane** przez wywołania funkcji zwracanej przez dekorator. Jak przekonałeś się już podczas przeglądania zamieszczonego na poprzedniej stronie rozwiązania, aby spełnić wymogi stawiane w punkcie 3. przepisu tworzenia dekoratora, zwróciliśmy opakowaną wersję istniejącej funkcji, która to wersja zawiera dodatkowy kod implementujący niezbędne działania. Ta opakowana wersja *dekoruje* istniejącą funkcję.

Pamiętaj, że konstrukcje *args oraz **kwargs obsługują dowolną liczbę i typ argumentów.

Mamy tu jednak pewien problem, ponieważ samo zapewnienie opakowania nie wystarczy; zachowane powinny być również *właściwości wywołania* dekorowanej funkcji. Oznacza to, że jeśli Twoja istniejąca funkcja przyjmowała dwa argumenty, funkcja opakowana również powinna to robić. Gdybyś z góry wiedział, ilu argumentów powinieneś się tu spodziewać, mógłbyś odpowiednio zaprojektować swój kod. Niestety nie wiesz tego wcześniej, ponieważ dekorator może zostać zastosowany do dowolnej istniejącej funkcji, która może mieć — i to jak najbardziej dosłownie — dowolną liczbę i typ argumentów.

Co powinieneś zatem zrobić? Rozwiązaniem jest zapewnienie jak największej „ogólności” i sprawienie, aby funkcja `wrapper` radziła sobie z obsługą dowolnej liczby i typu argumentów. Wiesz już, jak to zrobić, ponieważ widziałeś, co można osiągnąć za pomocą argumentów `*args` oraz `**kwargs`.

Zaostrz ołówek

Zmieńmy funkcję `wrapper` w taki sposób, aby przyjmowała dowolną liczbę i typ argumentów. Należy też zapewnić, aby funkcja `func`, gdy jest wywoływana, używała tej samej liczby i typu argumentów, jakie zostały przekazane funkcji `wrapper`. W pustych miejscach widocznych poniżej dodaj kod odpowiedzialny za argumenty.

```
def check_logged_in(func):
    def wrapper(                                  ):
        if 'logged_in' in session:
            return func(                          )
        return 'NIE jesteś zalogowany.'
    return wrapper
```

Co powinieneś dodać do sygnatury funkcji wrapper?

Zaostrz ołówek
Rozwiązanie

Miałeś zmienić funkcję `wrapper` w taki sposób, aby przyjmowała dowolną liczbę i typ argumentów, jak również zapewnić, aby funkcja `func`, gdy jest wywoływana, używała tej samej liczby i typu argumentów, jakie zostały przekazane funkcji `wrapper`.

Zastosowanie ogólnej sygnatury rozwiązuje problem, ponieważ zapewnia obsługę dowolnej liczby i typu argumentów. Zwróć uwagę, że funkcję func wywołujemy z tymi samym argumentami, które zostały podane funkcji wrapper, niezależnie od tego, jakie są to argumenty.

```
def check_logged_in(func):
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'NIE jesteś zalogowany.'
    return wrapper
```

Sprawa załatwiona... prawda?

Jeśli swoje wnioski wyciągasz na podstawie porównania tego, co zrobiliśmy, z tym, co znajduje się w naszym przepisie, wybaczymy Ci wiarę w to, że nasze zadanie dobiegło już końca. Dobiegło... prawie. Musimy sobie poradzić jeszcze z dwoma rzeczami: jedna związana jest ze wszystkimi dekoratorami, podczas gdy druga wiąże się z tym konkretnym dekoratorem.

Najpierw pozbądźmy się drugiego problemu, specyficznego dla tego przypadku. Jako że dekorator `check_logged_in` znajduje się w osobnym module, musimy zapewnić, aby wszystkie moduły, do których odwołujemy się w jego kodzie, były również zaimportowane w pliku *checker.py*. Dekorator ten wykorzystuje słownik `session`, który należy zaimportować z frameworka Flask, aby uniknąć błędów. Można to zrobić bardzo łatwo, ponieważ sprowadza się to do dodania przedstawionej poniżej instrukcji `import` na samym początku pliku modułu.

```
from flask import session
```

Pierwszy problem, który dotyczy *wszystkich* dekoratorów, jest związany ze sposobem, w jaki funkcje przedstawiają się interpreterowi. Po udekorowaniu i w sytuacji, gdy nie dołoży się odpowiednich starań, funkcja może „utracić" swoją tożsamość, co może z kolei prowadzić do kłopotów. Powód, dla którego tak się dzieje, ma bardzo techniczny charakter i jest dość egzotyczny, aby zaś go zrozumieć, trzeba dysponować wiedzą na temat działania wewnętrznych mechanizmów Pythona, której większość osób nie musi (ani nie chce) posiadać. W związku z tym biblioteka standardowa Pythona zapewnia moduł obsługujący odpowiednie szczegóły za Ciebie (dzięki czemu nigdy nie będziesz musiał się nimi martwić). Powinieneś jedynie pamiętać o zaimportowaniu niezbędnego modułu (o nazwie `functools`), a następnie wywołaniu jednej należącej do niego funkcji (o nazwie `wraps`).

Być może zakrawa to na pewną ironię, ale funkcja `wraps` jest zaimplementowana jako dekorator, dlatego tak naprawdę nie wywołujesz jej, lecz raczej używasz, aby udekorować swoją funkcję `wrapper` *wewnątrz* własnego dekoratora. Poszliśmy o krok dalej i zrobiliśmy to już za Ciebie. Uzupełniony kod dekoratora `check_logged_in` został zaprezentowany na początku następnej strony.

Tworząc własne dekoratory, zawsze importuj, a następnie stosuj funkcję wraps należącą do modułu functools.

Twój dekorator w pełni chwały

Zanim przejdziemy dalej, upewnij się, że kod Twojego dekoratora wygląda *dokładnie* tak samo, jak przedstawiony poniżej nasz.

Pamiętaj, aby zaimportować słownik session z modułu flask.

checker.py - C:/Users/lsuma/Desktop/Python/R10/checker...

File Edit Format Run Options Window Help

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'NIE jesteś zalogowany.'
    return wrapper
```

Ln: 13 Col: 0

Udekoruj funkcję wrapper dekoratorem wraps (pamiętając o przekazaniu mu func jako argumentu).

Zaimportuj funkcję wraps (która sama jest również dekoratorem) z modułu functools (który stanowi element składowy biblioteki standardowej).

Teraz, gdy moduł *checker.py* zawiera już ukończoną funkcję `check_logged_in`, wykorzystajmy go wewnątrz naszego programu *simple_webapp.py*. Poniżej znajdziesz bieżącą wersję kodu tej aplikacji WWW (którą pokazaliśmy tu w dwóch kolumnach, aby zaoszczędzić trochę cennego miejsca).

```
from flask import Flask, session

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Witaj, świecie, tu prosta aplikacja WWW.'

@app.route('/page1')
def page1():
    return 'To jest strona  1.'

@app.route('/page2')
def page2():
    return 'To jest strona  2.'

@app.route('/page3')
def page3():
    return 'To jest strona  3.'

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'Teraz jesteś zalogowany.'
```

```
@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'Teraz jesteś wylogowany.'

@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'W tej chwili jesteś zalogowany.'
    return 'NIE jesteś zalogowany.'

app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

Przypomnij sobie, że naszym celem jest tu ograniczenie dostępu do stron wskazywanych przez URL-e /page1, /page2 oraz /page3, które są obecnie dostępne dla wszystkich użytkowników (zgodnie z przedstawionym tu kodem).

Zaprzęganie Twojego dekoratora do pracy

Dostosowanie kodu programu *simple_webapp.py* do wykorzystania dekoratora `check_logged_in` nie jest szczególnie trudne. Oto lista czynności, które powinieneś wykonać:

1. **Zaimportuj dekorator.**
   Dekorator `check_logged_in` musi zostać zaimportowany z modułu *checker.py*. W tym celu należy tylko dodać wymaganą instrukcję `import` na samym początku kodu naszej aplikacji WWW.

2. **Usuń niepotrzebny kod.**
   Teraz, gdy istnieje już dekorator `check_logged_in`, nie jest nam więcej potrzebna funkcja `check_status`, dlatego możemy ją swobodnie usunąć z programu *simple_webapp.py*.

3. **Użyj dekoratora zgodnie z potrzebą.**
   Aby użyć dekoratora `check_logged_in`, zastosuj go do dowolnej funkcji naszej aplikacji WWW, korzystając ze składni ze znakiem `@`.

Poniżej raz jeszcze został przedstawiony kod programu *simple_webapp.py*, w którym dokonano trzech wymienionych powyżej zmian. Zwróć uwagę, że z URL-ami `/page1`, `/page2` oraz `/page3` związane są teraz dwa dekoratory: `@app.route` (zapewniany przez framework Flask) oraz `@check_logged_in` (który właśnie utworzyłeś).

```
from flask import Flask, session

from checker import check_logged_in

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Witaj, świecie, tu prosta aplikacja WWW.'

@app.route('/page1')
@check_logged_in
def page1():
    return 'To jest strona  1.'

@app.route('/page2')
@check_logged_in
def page2():
    return 'To jest strona  2.'

@app.route('/page3')
@check_logged_in
def page3():
    return 'To jest strona  3.'

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'Teraz jesteś zalogowany.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'Teraz jesteś wylogowany.'

app.secret_key =
'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

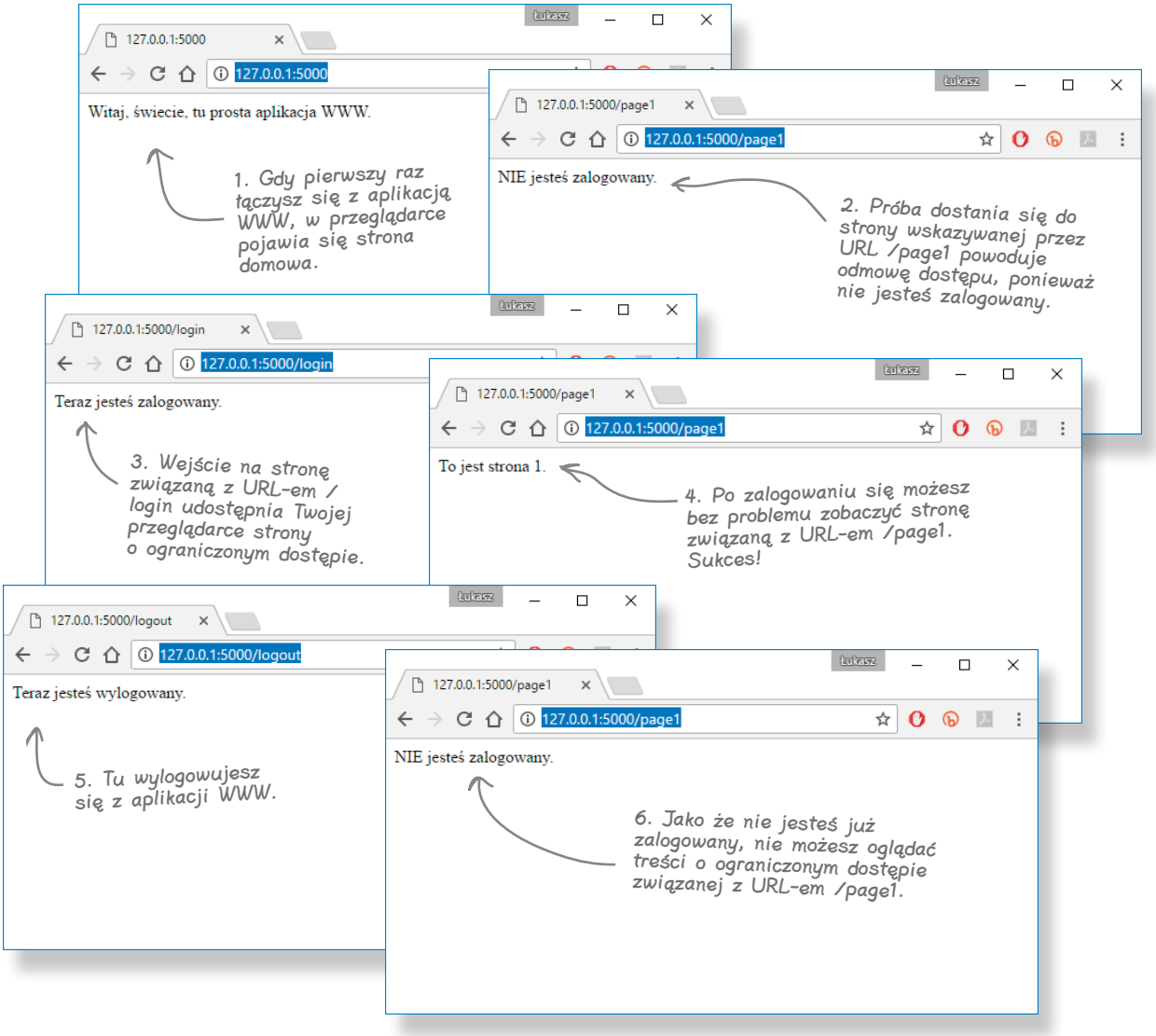
Zastosuj operator do istniejącej funkcji, korzystając ze składni ze znakiem @.

Nie zapomnij wprowadzić wyróżnionych zmian w swojej aplikacji WWW, ZANIM przejdziesz dalej.

Jazda próbna

Aby przekonać się, czy nasz sprawdzający zalogowanie dekorator działa tak, jak sobie tego życzymy, weźmy w obroty wykorzystującą go wersję aplikacji *simple_webapp.py*.

Uruchomiwszy aplikację WWW, skorzystaj z przeglądarki internetowej, aby spróbować dostać się na stronę związaną z URL-em `/page1` przed zalogowaniem się. Po zalogowaniu się spróbuj ponownie otworzyć tę stronę, a następnie, po wylogowaniu, spróbuj raz jeszcze odwołać się do zawartości o ograniczonym dostępie. Sprawdźmy, co się stanie.



Piękno dekoratorów

Przyjrzyj się raz jeszcze przedstawionemu poniżej kodowi swojego dekoratora `check_logged_in`. Zauważ, jak wydziela on logikę używaną do sprawdzania, czy użytkownik przeglądarki jest zalogowany: umieszcza ten (potencjalnie skomplikowany) kod w jednym miejscu — *wewnątrz* dekoratora — a następnie udostępnia go w całym programie, a wszystko to dzięki zastosowaniu składni dekoratora `@check_logged_in`.

checker.py - C:/Users/lsuma/Desktop/Python/R10/checker...

File Edit Format Run Options Window Help

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'NIE jesteś zalogowany.'
    return wrapper
```

Ln: 13 Col: 0

Ten kod wygląda dziwnie, ale tak naprawdę wszystko z nim w porządku.

Wyodrębnienie kodu w dekoratorze sprawia, że używający go kod staje się czytelniejszy. Zobacz poniżej, jak wygląda zastosowanie naszego dekoratora w przypadku kodu obsługi URL-a `/page2`.

simple_webapp3.py - C:/Users/lsuma/Desktop/Python/R10/simple...

File Edit Format Run Options Window Help

```
@app.route('/page2')
@check_logged_in
def page2():
    return 'To jest strona  2.'
```

Ln: 23 Col: 0

Zastosowanie dekoratora sprawia, że ten kod znacznie łatwiej się czyta.

Dekoratory nie są dziwaczne; są świetne.

Zwróć uwagę, że kod funkcji `page2` dotyczy już wyłącznie tego, czego powinien, a więc odpowiada za wyświetlenie treści związanej z URL-em `/page2`. W naszym przykładzie kod ten składa się z pojedynczej, prostej instrukcji; byłby jednak mniej czytelny i trudniejszy do zrozumienia, gdyby zawierał *również* logikę niezbędną do sprawdzenia, czy użytkownik przeglądarki jest zalogowany, czy nie. Użycie dekoratora w celu odseparowania kodu sprawdzania zalogowania stanowi doskonałe rozwiązanie.

To „wyodrębnienie logiki" stanowi jeden z powodów, dla których zastosowanie dekoratorów jest popularne wśród programistów Pythona. Innym jest fakt, że — gdy dobrze się nad tym zastanowić — tworząc dekorator `check_logged_in`, udało Ci się utworzyć kod, który *rozszerza istniejącą funkcję o dodatkowe możliwości, zmieniając sposób jej działania, lecz nie modyfikując jej kodu*. Gdy przedstawiliśmy Ci tę ideę po raz pierwszy w tym rozdziale, określiliśmy ją jako „dziwną". Teraz jednak, gdy zastosowałeś to rozwiązanie w praktyce, chyba nie wydaje Ci się już ono takie, prawda?

Tworzenie kolejnych dekoratorów

Po zakończeniu procedury tworzenia dekoratora `check_logged_in` możesz skorzystać z jego kodu jako podstawy wszystkich nowych dekoratorów, które zamierzasz opracować w przyszłości.

Aby ułatwić Ci to zadanie, poniżej przedstawiamy ogólny szablon kodu (znajdujący się w pliku o nazwie *tmpl_decorator.py*), którego możesz używać jako bazy dla nowych, tworzonych przez siebie dekoratorów.

tmpl_decorator.py - C:\Users\lsuma\Desktop\Python\R10\tmpl_decorator.py (3.6.0)

File Edit Format Run Options Window Help

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Kod, który ma być wykonany PRZED wywołaniem dekorowanej funkcji.

        # 2. Wywołanie dekorowanej funkcji zgodnie z potrzebą,
        #    zwrócenie jej wyniku, jeśli to konieczne.
             return func(*args, **kwargs)

        # 3. Kod, który ma być wykonany ZAMIAST wywołania dekorowanej funkcji.
    return wrapper
```

Ln: 15 Col: 0

Zastąp te komentarze kodem nowego dekoratora.

Ten szablon kodu można odpowiednio dostosować do swoich potrzeb. Musisz jedynie nadać nowemu dekoratorowi właściwą nazwę, a następnie zastąpić trzy komentarze widoczne w jego zestawie kodem specyficznym dla zadania, które ma wykonać.

Jeśli wywołanie dekorowanej funkcji bez zwracania jej wartości ma w przypadku Twojego nowego dekoratora sens, to w porządku. W końcu wszystko, co umieszczasz w funkcji `wrapper`, jest Twoim kodem i wolno Ci tu robić, cokolwiek zechcesz.

Nie istnieją głupie pytania

P: Czy dekoratory nie przypominają menedżera kontekstu, o którym była mowa w poprzednim rozdziale, w tym sensie, że obydwie te konstrukcje umożliwiają opakowywanie kodu w dodatkowe możliwości?

O: To świetne pytanie. Odpowiedź brzmi: tak *i* nie. Tak, bo zarówno dekoratory, jak i menedżery kontekstu umożliwiają rozszerzanie istniejącego kodu o dodatkową logikę. Lecz nie, nie są takie same. Zadaniem dekoratorów jest konkretnie rozszerzanie istniejących funkcji o dodatkowe możliwości, podczas gdy w przypadku menedżerów kontekstu chodzi o zapewnienie, aby kod był wykonywany w określonym kontekście, co realizuje się przez wykonanie pewnych operacji przed instrukcją `with` oraz wymuszenie, że jakieś operacje zostaną **zawsze** wykonane po niej. Podobną rzecz możesz osiągnąć za pomocą dekoratorów, ale większość programistów Pythona uznałaby Cię za osobę trochę szaloną, gdybyś spróbował to zrobić. Zauważ również, że kod dekoratora nie ma obowiązku robić czegokolwiek po wywołaniu dekorowanej funkcji (jak ma to miejsce w przypadku dekoratora `check_logged_in`, który nie wykonuje tu żadnej operacji). To zachowanie dekoratora zasadniczo różni się od protokołu, zgodnie z którym powinny działać menedżery kontekstu.

Powrót do ograniczania dostępu do strony /viewlog

Gdy opracowałeś już mechanizm umożliwiający Ci ograniczenie dostępu do określonych URL-i w aplikacji *simple_webapp.py*, zastosowanie tego samego mechanizmu w dowolnej innej aplikacji to łatwizna.

Dotyczy to również programu *vsearch4web.py*, w którym chcieliśmy ograniczyć dostęp do strony wskazywanej przez URL `/viewlog`. Musisz tu tylko skopiować funkcje `do_login` oraz `do_logout` z pliku *simple_webapp.py* do pliku *vsearch4web.py*, zaimportować moduł z pliku *checker.py*, a następnie udekorować funkcję `view_the_log` za pomocą dekoratora `check_logged_in`. Zgoda, z pewnością zechcesz dodać trochę zmian w funkcjach `do_login` oraz `do_logout` (na przykład porównując w nich podane przez użytkownika dane logowania z tymi, które są przechowywane w bazie danych), ale kwestię ograniczania dostępu do określonych URL-i w dużej mierze załatwia Ci już sam dekorator `check_logged_in`.

Co dalej?

Zamiast poświęcać kilka kolejnych stron na wprowadzanie do programu *vsearch4web.py* zmian, których właśnie dokonaliśmy w kodzie aplikacji *simple_webapp.py*, pozostawimy Ci to zadanie do *samodzielnego* wykonania. Na początku następnego rozdziału zaprezentujemy zaktualizowaną wersję kodu programu *vsearch4web.py*, dzięki czemu będziesz mógł porównać go ze swoim, ponieważ kod ten będzie wykorzystywany w treści tego rozdziału.

Do tej pory wszystkie przykłady kodu zaprezentowane w niniejszej książce powstawały przy założeniu, że nic złego nie może się zdarzyć i nic nigdy nie idzie źle. Wybór tej strategii był z naszej strony celowy, ponieważ chcieliśmy, abyś dobrze poznał Pythona, zanim wgłębimy się w tematy związane z poprawianiem błędów, ich unikaniem i wykrywaniem, obsługą wyjątków i tym podobnymi zagadnieniami.

Dotarliśmy właśnie do punktu, w którym nie możemy już dalej trzymać się tej strategii. Środowiska, w których wykonywany jest nasz kod, są prawdziwe i rzeczy mogą iść nie po naszej myśli (co więcej, często tak się dzieje). Niektóre problemy można naprawić (lub ich uniknąć), lecz niektórych nie. Jeśli to tylko możliwe, będziesz chciał, aby Twój kod radził sobie z większością sytuacji awaryjnych, doprowadzając do przerwania działania programu tylko wówczas, gdy wydarzy się coś naprawdę wyjątkowego, co będzie poza Twoją kontrolą. W następnym rozdziale przyjrzymy się różnym strategiom określania, co należy zrobić, gdy sprawy idą w złym kierunku.

Zanim jednak do tego przejdziemy, przejrzyj najważniejsze zagadnienia, które zostały poruszone w niniejszym rozdziale.

CELNE SPOSTRZEŻENIA

- Gdy w aplikacji WWW wykorzystującej framework Flask zachodzi potrzeba przechowywania stanu po stronie serwera, skorzystaj ze słownika `session` (i nie zapomnij tu ustawić trudną do odgadnięcia wartość atrybutu `secret_key`).
- Funkcję można przesłać jako argument wywołania innej funkcji. Użycie samej nazwy funkcji (bez nawiasów) umożliwia uzyskanie **obiektu funkcji**, którym można się posługiwać tak jak każdą inną zmienną.
- Gdy obiekt funkcji przekazujesz jako argument innej funkcji, w kodzie funkcji odbierającej możesz **wywołać** otrzymany obiekt funkcji, dołączając do jego nazwy nawias.
- Funkcja może być **zagnieżdżona** w zestawie innej (obejmującej) funkcji (jest wówczas widoczna tylko w zasięgu tej obejmującej funkcji).
- Poza tym, że funkcje mogą przyjmować obiekty funkcji jako argumenty, mogą też **zwracać** zagnieżdżone funkcje jako swoje wartości zwracane.
- `*args` to skrót oznaczający „rozwinięcie do postaci listy elementów".
- `**kwargs` to skrót oznaczający „rozwinięcie do postaci słownika składającego się z kluczy i wartości". Gdy widzisz litery „kw", pomyśl o „słowach kluczowych" (ang. *keywords*).
- Zarówno zapis *, jak i ** mogą być wykorzystywane w wywołaniach funkcji, dzięki czemu lista lub kolekcja słów kluczowych może być przekazywana do funkcji jako pojedynczy (rozszerzalny) argument.
- Zastosowanie konstrukcji `(*args, **kwargs)` w roli **sygnatury funkcji** umożliwia tworzenie funkcji, które przyjmują dowolną liczbę i typ argumentów.
- Korzystając z nowych, przedstawionych w tym rozdziale możliwości związanych z funkcjami, nauczyłeś się tworzyć **dekorator funkcji**, który zmienia zachowanie istniejącej funkcji bez konieczności modyfikowania kodu samej funkcji. Brzmi to dość dziwacznie, ale jest świetną sprawą (jak również bardzo użyteczną).

Kod przykładów z rozdziału 10. (1. z 2)

To jest zawartość pliku quick_session.py.

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'NigdyNieZgadniesz'


@app.route('/setuser/<user>')
def setuser(user: str) -> str:
   session['user'] = user
   return 'Użytkownikiem jest: ' + session['user']


@app.route('/getuser')
def getuser() -> str:
   return 'Użytkownikiem jest obecnie: ' + session['user']


if __name__ == '__main__':
   app.run(debug=True)
```

To jest zawartość pliku checker.py, w którym znajduje się kod tworzonego w tym rozdziale dekoratora check_logged_in.

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'NIE jesteś zalogowany.'
    return wrapper
```

To jest zawartość pliku tmpl_decorator.py, stanowiącego wygodny szablon do tworzenia dekoratorów, z którego możesz korzystać, gdy tylko poczujesz, że jest taka potrzeba.

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Kod, który ma być wykonany PRZED wywołaniem dekorowanej funkcji.

        # 2. Wywołanie dekorowanej funkcji zgodnie z potrzebą,
        #    zwrócenie jej wyniku, jeśli to konieczne.
             return func(*args, **kwargs)

        # 3. Kod, który ma być wykonany ZAMIAST wywołania dekorowanej funkcji.
    return wrapper
```

Kod przykładów z rozdziału 10. (2. z 2)

```
from flask import Flask, session

from checker import check_logged_in

app = Flask(__name__)


@app.route('/')
def hello():
    return 'Witaj, świecie, tu prosta aplikacja WWW.'


@app.route('/page1')
@check_logged_in
def page1():
    return 'To jest strona  1.'


@app.route('/page2')
@check_logged_in
def page2():
    return 'To jest strona  2.'


@app.route('/page3')
@check_logged_in
def page3():
    return 'To jest strona  3.'


@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'Teraz jesteś zalogowany.'


@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'Teraz jesteś wylogowany.'


app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

To jest zawartość pliku simple_webapp.py, który łączy w sobie cały kod przedstawiony w niniejszym rozdziale. Gdy zachodzi konieczność ograniczenia dostępu do określonych URL-i, oprzyj swoją strategię na przedstawionym tu mechanizmie, który został wykorzystany w aplikacji WWW.

Użycie dekoratorów ułatwia naszym zdaniem czytanie i zrozumienie kodu tej aplikacji. Nie sądzisz? :)

Co robić, gdy sprawy idą źle

Sprawy idą źle, dzieje się tak zawsze — niezależnie od tego, jak dobry jest Twój kod.

Jak do tej pory wszystkie przykłady przedstawione w tej książce bez problemu udawało Ci się uruchomić i raczej jesteś pewien, że zaprezentowany dotąd kod działa. Czy oznacza to jednak, że kod ten jest solidny? Raczej nie. Pisanie kodu w oparciu o założenie, że nic złego nie może się zdarzyć, jest (w najlepszym przypadku) oznaką naiwności. W najgorszym przypadku jest to wręcz praktyka niebezpieczna, ponieważ nieprzewidziane rzeczy się zdarzają (i będą się zdarzać). Lepiej, abyś tworząc kod, był ostrożny niż beztroski. Ostrożność jest konieczna, aby mieć pewność, że Twój kod robi to, czego od niego oczekujesz, jak również że reaguje prawidłowo, gdy rzeczy idą w złą stronę. W niniejszym rozdziale dowiesz się nie tylko tego, co może pójść nie tak, lecz również co robić, gdy to się stanie (a częstokroć jeszcze zanim to nastąpi).

Długie ćwiczenie

Zaczniemy ten rozdział od skoku na głęboką wodę. Poniżej została zaprezentowana ostatnia wersja kodu aplikacji WWW *vsearch4web.py*. Jak możesz się przekonać, zaktualizowaliśmy ten kod w taki sposób, aby korzystał z dekoratora `check_logged_in`, który został przedstawiony w poprzednim rozdziale, w celu kontrolowania, kiedy informacje prezentowane na stronie związanej z URL-em `/viewlog` są (lub nie są) widoczne dla użytkowników.

Poświęć tyle czasu, ile tylko Ci trzeba, aby przeczytać ten kod, a następnie użyj ołówka, aby zakreślić i skomentować te części kodu, które Twoim zdaniem mogą powodować problemy podczas działania w środowisku produkcyjnym. Wskaż *wszystko*, co — jak sądzisz — może być przyczyną kłopotów, nie jedynie potencjalne problemy czy błędy czasu wykonania.

```
from flask import Flask, render_template, request, escape, session
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
   session['logged_in'] = True
   return 'Teraz jesteś zalogowany.'

@app.route('/logout')
def do_logout() -> str:
   session.pop('logged_in')
   return 'Teraz jesteś wylogowany.'

def log_request(req: 'flask_request', res: str) -> None:
   with UseDatabase(app.config['dbconfig']) as cursor:
      _SQL = """insert into log
                (phrase, letters, ip, browser_string, results)
                values
                (%s, %s, %s, %s, %s)"""
      cursor.execute(_SQL, (req.form['phrase'],
                            req.form['letters'],
                            req.remote_addr,
                            req.user_agent.browser,
                            res, ))
```

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   log_request(request, results)
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej
search4letters!')

@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
   with UseDatabase(app.config['dbconfig']) as cursor:
      _SQL = """select phrase, letters, ip, browser_string, results
                from log"""
      cursor.execute(_SQL)
      contents = cursor.fetchall()
   titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
   return render_template('viewlog.html',
                          the_title='View Log',
                          the_row_titles=titles,
                          the_data=contents,)

app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

Długie ćwiczenie Rozwiązanie

Miałeś poświęcić tyle czasu, ile tylko było trzeba, aby przeczytać przedstawiony poniżej kod (będący zaktualizowaną wersją kodu aplikacji WWW *vsearch4web.py*). Następnie, używając ołówka, miałeś zakreślić i skomentować te jego części, które Twoim zdaniem mogą powodować problemy podczas działania w środowisku produkcyjnym. Powinieneś był wskazać *wszystko*, co — jak sądzisz — może być przyczyną kłopotów, nie jedynie potencjalne problemy czy błędy czasu wykonania. (Ponumerowaliśmy nasze komentarze, aby łatwiej było się do nich odwoływać).

```
from flask import Flask, render_template, request, escape, session
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'Teraz jesteś zalogowany.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'Teraz jesteś wylogowany.'

def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

1. Co się stanie, gdy połączenie z bazą danych się nie powiedzie?

2. Czy te instrukcje SQL są chronione przed paskudnymi, wykorzystującymi sieć atakami, takimi jak wstrzyknięcie kodu SQL (ang. SQL injection) lub wstrzyknięcie skryptu (ang. Cross-site scripting)?

3. Co się stanie, gdy wykonanie tych instrukcji SQL będzie trwało zbyt długo?

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   log_request(request, results)
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
   return render_template('entry.html',
                          the_title='Witamy na stronie internetowej search4letters!')

@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
   with UseDatabase(app.config['dbconfig']) as cursor:
      _SQL = """select phrase, letters, ip, browser_string, results
                from log"""
      cursor.execute(_SQL)
      contents = cursor.fetchall()
   titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
   return render_template('viewlog.html',
                          the_title='View Log',
                          the_row_titles=titles,
                          the_data=contents,)

app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

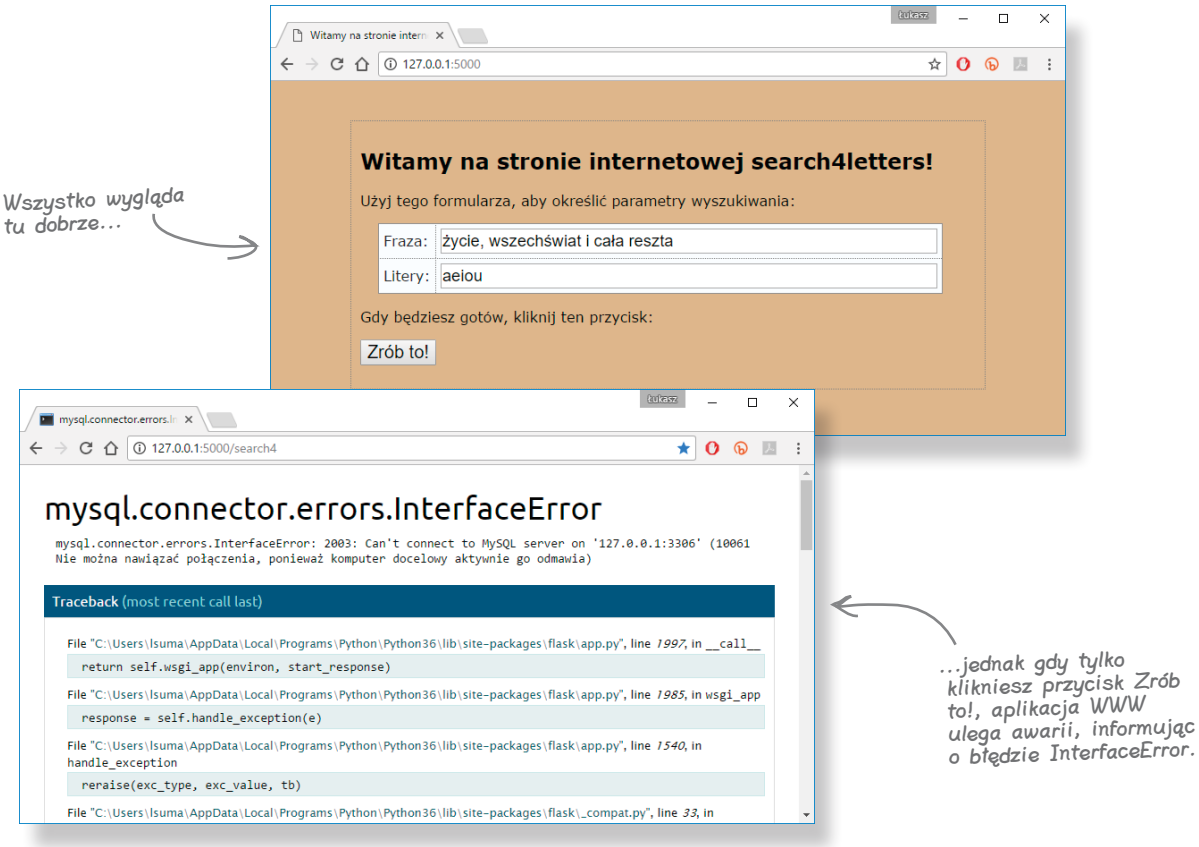
4. Co się stanie, gdy ta operacja się nie uda?

Bazy danych nie są zawsze dostępne

W kodzie aplikacji *vsearch4web.py* wskazaliśmy cztery potencjalne problemy i przyznajemy, że może ich tam być znacznie więcej, lecz na razie zajmiemy się tylko tymi, które znaleźliśmy. Każdemu z nich przyjrzyjmy się dokładniej (zrobimy to na tej i kilku kolejnych stronach, po prostu opisując poszczególne problemy; *ich rozwiązywaniem zajmiemy się w dalszej części niniejszego rozdziału*). Nasze pierwsze zmartwienie dotyczy dostępu do bazy danych.

1. **Co się stanie, gdy połączenie z bazą danych się nie powiedzie?** Nasza aplikacja WWW zachowuje się, jakby niewinnie zakładała, że używana przez nią baza danych jest zawsze dostępna i sprawna, może tak jednak nie być (z wielu różnych powodów). W tej chwili nie jest do końca jasne, co się stanie, gdy baza danych nie będzie działać, ponieważ nasz kod zupełnie nie jest przygotowany na taką ewentualność.

Sprawdźmy, co się wydarzy, gdy tymczasowo *wyłączymy* naszą bazę danych. Jak możesz zobaczyć poniżej, nasza aplikacja WWW ładuje się bez problemu, lecz gdy tylko spróbujemy coś zrobić, w oknie przeglądarki pojawi się zniechęcający komunikat błędu.



Wszystko wygląda tu dobrze...

...jednak gdy tylko klikniesz przycisk Zrób to!, aplikacja WWW ulega awarii, informując o błędzie InterfaceError.

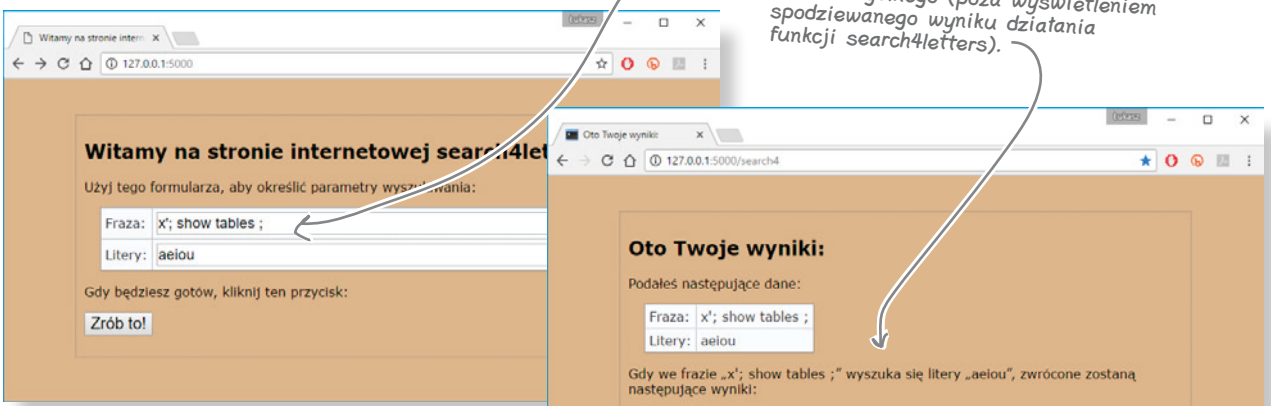
Ataki sieciowe to prawdziwy problem

Oprócz ewentualnymi kłopotami z dostępnością bazy danych powinniśmy się również martwić paskudnymi osobami próbującymi robić paskudne rzeczy Twojej aplikacji WWW, co doprowadza nas do drugiego problemu.

2 **Czy Twoja aplikacja WWW jest chroniona przed atakami sieciowymi?**
Określenia *wstrzyknięcie kodu SQL* (ang. *SQL injection*, *SQLi*) oraz *wstrzyknięcie skryptu* (ang. *Cross-site scripting*, *XSS*) powinny zasiać strach w sercu każdego programisty WWW. Pierwszy z wymienionych tu rodzajów ataku umożliwia napastnikom włamanie się do Twojej bazy danych, drugi zaś — włamanie się do serwisu WWW. Istnieją również inne rodzaje włamań sieciowych, którymi powinieneś się martwić, ale wspomniane tu stanowią „wielką dwójkę".

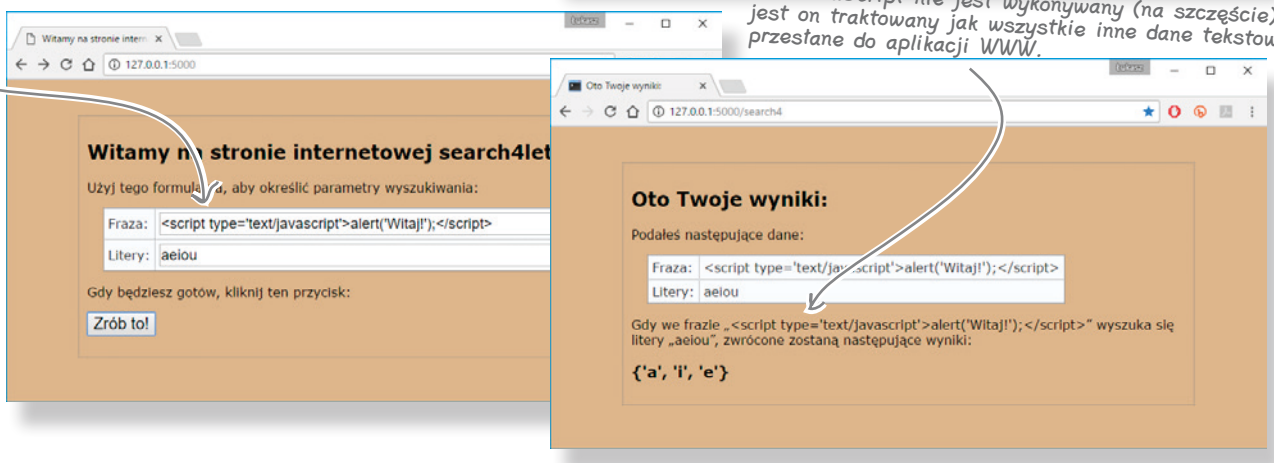
Podobnie jak przy pierwszym problemie, przekonajmy się, co będzie, gdy spróbujemy zasymulować te włamania w przypadku Twojej aplikacji WWW. Jak możesz zobaczyć poniżej, wygląda na to, że jesteśmy przygotowani na odparcie obydwu ataków.

Gdy spróbujesz wstrzyknąć kod SQL do interfejsu WWW, nie wydarzy się nic szczególnego (poza wyświetleniem spodziewanego wyniku działania funkcji search4letters).



Wszelkie próby włamania XSS polegające na wprowadzeniu kodu JavaScript do aplikacji WWW nie odnoszą skutku.

Kod JavaScript nie jest wykonywany (na szczęście); jest on traktowany jak wszystkie inne dane tekstowe przesłane do aplikacji WWW.



Operacje wejścia-wyjścia są (czasami) powolne

W tej chwili nasza aplikacja WWW komunikuje się z bazą danych działającą po stronie backendu niemal natychmiastowo, dzięki czemu jej użytkownicy obserwują bardzo niewielkie opóźnienia wynikające z interakcji zachodzących pomiędzy aplikacją a bazą lub wręcz nie ma takich opóźnień. Wyobraź sobie jednak sytuację, w której komunikacja z bazą danych wymaga pewnego czasu, być może liczonego nawet w sekundach.

3 **Co się stanie, gdy coś będzie trwało zbyt długo?**
Być może baza danych pracująca po stronie backendu znajduje się na innej maszynie, w innym budynku lub nawet na innym kontynencie... Co się wtedy stanie?

Komunikacja z bazą danych działającą po stronie backendu może wymagać czasu. W gruncie rzeczy zawsze, gdy zachodzi jakaś interakcja pomiędzy Twoim kodem a czymś, co jest wobec niego zewnętrzne (na przykład: plikiem, bazą danych, siecią czy jakimkolwiek innym zasobem tego rodzaju), interakcja ta może wymagać dowolnie długiego czasu, na którego ilość zwykle nie masz żadnego wpływu. Mimo tego braku kontroli musisz być świadomy, że niektóre operacje mogą trwać długo.

Aby zademonstrować ten problem, dodajmy do naszej aplikacji WWW *sztuczne* opóźnienie (korzystając w tym celu z funkcji `sleep`, która jest częścią składową modułu `time` należącego do biblioteki standardowej). Dodaj przedstawiony poniżej wiersz na samym początku kodu Twojej aplikacji WWW (w okolicach innych instrukcji `import`).

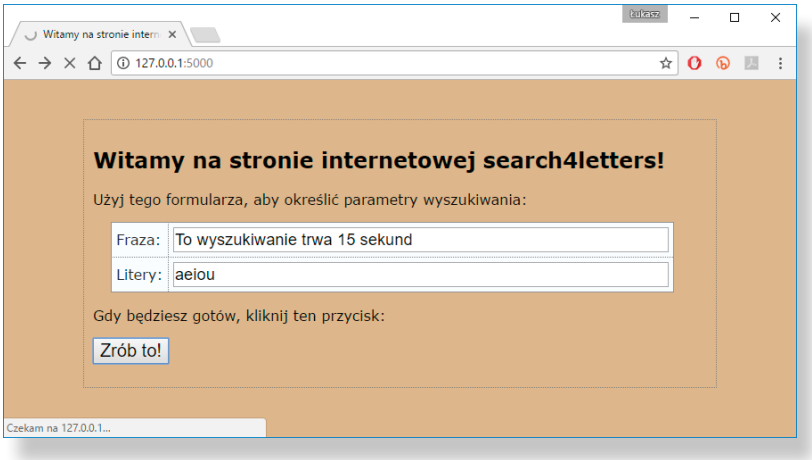
```
from time import sleep
```

Po wstawieniu tej instrukcji przejdź do edycji funkcji `log_request` i umieść w niej pokazany poniżej wiersz kodu tuż przed instrukcją `with`.

```
sleep(15)
```

Gdy zrestartujesz swoją aplikację WWW, a następnie rozpoczniesz wyszukiwanie, zaobserwujesz znaczące opóźnienie związane z tym, że przeglądarka internetowa będzie oczekiwała na odpowiedź od Twojej aplikacji. W porównaniu z opóźnieniami spotykanymi we współczesnych sieciach 15 sekund będzie się wydawało całą wiecznością, co prawdopodobnie spowoduje, że użytkownicy Twojej aplikacji WWW pomyślą, iż uległa ona *awarii*.

Gdy klikniesz przycisk Zrób to!, Twoja przeglądarka internetowa będzie czekać... i czekać... i czekać... i czekać...

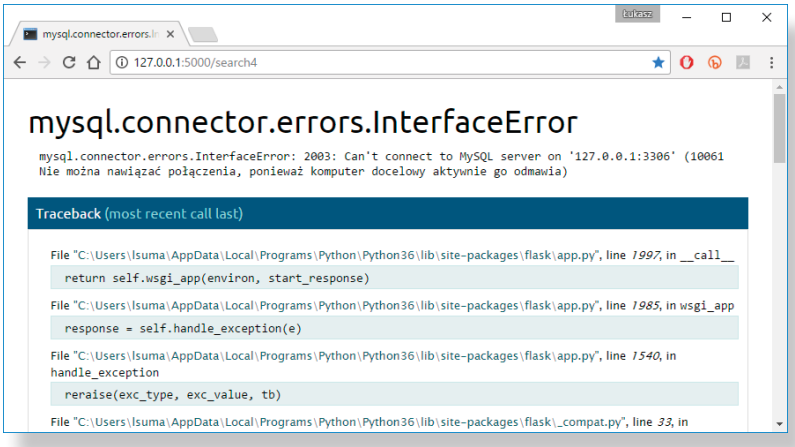


Wywołania Twoich funkcji mogą się nie powieść

Ostatni problem, który udało nam się zidentyfikować w ramach ćwiczenia rozpoczynającego ten rozdział, ma związek z wywołaniem funkcji `log_request` w obrębie funkcji `do_search`.

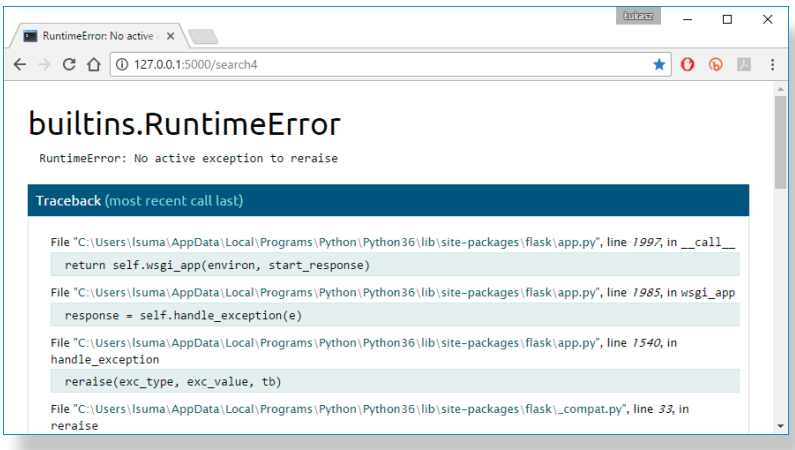
4 **Co się stanie, gdy nie powiedzie się wywołanie funkcji?**
Nigdy nie ma gwarancji, że wywołanie funkcji się powiedzie, zwłaszcza gdy funkcja ta wchodzi w interakcję z czymś zewnętrznym w stosunku do Twojego kodu.

Widzieliśmy już, co może się stać, gdy baza danych działająca po stronie backendu jest niedostępna — aplikacja WWW ulega awarii, zgłaszając błąd `InterfaceError`.



Nie ma bazy danych, dlatego Twoja aplikacja WWW ulega awarii.

Mogą się tu pojawić również inne problemy. Aby zasymulować kolejny błąd, w kodzie swojej aplikacji WWW znajdź dodaną przed chwilą (przy okazji omawiania punktu 3. naszej listy problemów) instrukcję `sleep(15)`, a następne zastąp ją pojedynczą instrukcją `raise`. Próba wykonania tej instrukcji przez interpreter powoduje wymuszenie wystąpienia błędu czasu wykonania. Gdy ponownie uruchomisz swoją aplikację WWW, Twoim oczom ukaże się *inny* komunikat błędu, co pokazano poniżej.



Coś innego poszło nie tak, a Twoja aplikacja WWW ponownie uległa awarii.

Zanim przejdziesz do następnej strony, usuń instrukcję `raise` ze swojego kodu, aby ponownie zapewnić aplikacji normalne działanie.

Rozważenie zidentyfikowanych problemów

Udało nam się wskazać cztery problemy dotyczące kodu aplikacji *vsearch4web.py*. Przyjrzyjmy się każdemu z nich raz jeszcze i zastanówmy się nad kolejnymi krokami.

1. Zawodzi połączenie z bazą danych

Błędy pojawiają się, gdy zewnętrzny system, od którego w jakiś sposób zależy działanie Twojego kodu, staje się niedostępny. W takich sytuacjach, z których przykładem mieliśmy do czynienia na poprzednich stronach, interpreter zgłosił wystąpienie błędu `InterfaceError`. Korzystając z wbudowanego mechanizmu obsługi wyjątków Pythona, można wykryć tego rodzaju błędy, a także odpowiednio na nie zareagować. A skoro można to zrobić, to zdecydowanie trzeba.

2. Twoja aplikacja jest celem ataków

Choć niektórzy mogą twierdzić, że troska o zabezpieczenie przed atakami na Twoją aplikację należy do zmartwień programistów WWW, wszelkie sposoby poprawiające odporność kodu, który piszesz, zawsze warte są rozważenia. W przypadku aplikacji *vsearch4web.py* ochrona przed „wielką dwójką" wektorów ataku, czyli *wstrzyknięciem kodu SQL* (*SQLi*) oraz *wstrzyknięciem skryptu* (*XSS*), jest już zapewniona. To raczej kwestia szczęśliwego trafu niż celowego działania z naszej strony, ponieważ biblioteka Jinja2 została zbudowana w taki sposób, aby z definicji być odporną na próby ataków *XSS* dzięki temu, że wszystkie potencjalnie problematyczne łańcuchy znakowe są w jej przypadku przetwarzane za pomocą odpowiednich znaków ucieczki (przypomnij sobie, że kod JavaScript, którym próbowaliśmy oszukać naszą aplikację WWW, nie został wykonany i nie miał żadnego wpływu na jej działanie). Jeśli zaś chodzi o ataki *SQLi*, to w wyniku zastosowania przez nas parametryzowanych łańcuchów znaków SQL obsługiwanych przez DB-API (i zawierających wszystkie te znaki zastępcze ?) zapewniliśmy — również dzięki sposobowi, w jaki zostały opracowane odpowiednie moduły — że nasz kod jest zabezpieczony przed całą klasą potencjalnych ataków.

Porady maniaka

Jeśli chcesz się dowiedzieć więcej na temat ataków *SQLi* oraz *XSS*, to doskonałym miejscem do rozpoczęcia poszukiwań odpowiednich informacji jest Wikipedia. W tym celu odwiedź strony *https://pl.wikipedia.org/wiki/SQL_injection* oraz *https://pl.wikipedia.org/wiki/Cross-site_scripting*. Pamiętaj też jednak, że istnieje wiele innych rodzajów ataków, które mogą sprawić sporo kłopotów Twojej aplikacji. Wymienione tutaj stanowią tylko „wielką dwójkę".

3. Wykonanie Twojego kodu wymaga dużo czasu

Jeśli wykonanie Twojego kodu trwa długo, musisz wziąć pod uwagę wrażenie, jakie ma w związku z tym użytkownik Twojej aplikacji. Jeśli nie zauważa opóźnienia, to prawdopodobnie wszystko jest w porządku. Jeśli jednak musi czekać, raczej będziesz zmuszony coś z tym zrobić (w przeciwnym razie użytkownik może stwierdzić, że gra nie jest warta świeczki i pójdzie sobie gdzieś indziej).

4. Zawodzą wywołania Twoich funkcji

Nie tylko zewnętrzne systemy mogą generować wyjątki w czasie pracy interpretera, może to robić również Twój kod. Gdy się to dzieje, musisz być gotów do wykrycia wyjątku i odpowiedniej reakcji na jego wystąpienia. Mechanizm, z którego w takim przypadku należy skorzystać, jest tym samym, o którym wspomnieliśmy przy okazji omawiania punktu 1. zaprezentowanego powyżej.

A więc... od czego powinniśmy *zacząć* w kwestii obsługi tych czterech wymienionych tu problemów? Z pierwszym i czwartym z nich możemy sobie poradzić za pomocą tego samego mechanizmu, a więc zaczniemy właśnie od tego.

Kod podatny na błędy powinieneś zawsze próbować wykonywać

Gdy coś idzie nie tak w związku z działaniem Twojego kodu, Python zgłasza wyjątek **czasu wykonania**. Wyjątek możesz traktować jak kontrolowaną awarię programu wywołaną przez interpreter.

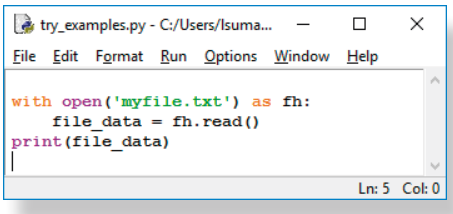
Jak miałeś okazję się przekonać na przykładach punktów 1. i 4., wyjątki mogą być zgłaszane w związku z wieloma różnymi okolicznościami. W rzeczywistości interpreter zapewnia całkiem sporo wbudowanych typów wyjątków, z których `RuntimeError` (wspomniany w punkcie 4.) stanowi tylko jeden z przykładów. Oprócz korzystania z wbudowanych typów wyjątków możliwe jest też definiowanie własnych wyjątków. Miałeś już nawet okazję spotkać się z jednym ich przykładem, którym był `InterfaceError` (wspomniany w punkcie 1.), zdefiniowany w ramach modułu *MySQL Connector*.

Pełną listę wbudowanych wyjątków znajdziesz pod adresem *https://docs.python.org/3/library/exceptions.html.*

Aby wykryć (i, o ile się uda, odpowiednio obsłużyć) wyjątek czasu wykonania, skorzystaj z instrukcji `try` języka Python, która pomaga zarządzać wyjątkami występującymi w czasie wykonania programu.

Aby wypróbować tę instrukcję w praktyce, zajmijmy się najpierw krótkim fragmentem kodu, który może ulec awarii w czasie wykonania. Poniżej znajdziesz trzy wyglądające całkiem niewinnie, lecz potencjalnie sprawiające problemy, wiersze kodu, którym powinieneś się dokładnie przyjrzeć.

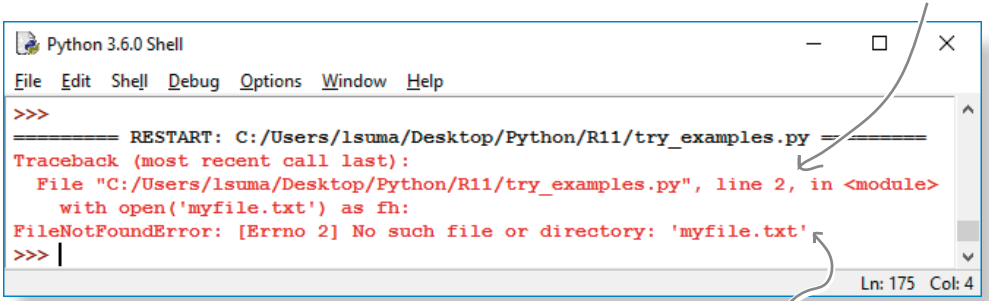
W kodzie tym nie ma nic dziwnego ani nadzwyczajnego: plik o określonej nazwie zostaje otwarty, a zapisane w nim dane są odczytywane i następnie wyświetlane na ekranie.



Z tymi trzema wierszami kodu wszystko jest absolutnie w porządku i — w formie, którą aktualnie mają — bez problemu się wykonają. Kod ten może jednak ulec awarii, jeśli nie będzie w stanie uzyskać dostępu do pliku *myfile.txt*. Może to być spowodowane brakiem tego pliku lub brakiem uprawnień Twojego kodu do odczytu danych. Jeśli kod ten zawiedzie, zostanie zgłoszony wyjątek, tak jak zostało to pokazane poniżej.

Gdy pojawia się błąd czasu wykonania, Python wyświetla „ślad" (ang. traceback) zawierający szczegółowy opis tego, co poszło nie tak i gdzie do tego doszło. W tym przypadku interpreter podejrzewa wiersz 2.

Ups!



Choć wygląda dość paskudnie, komunikat tracebacku jest bardzo przydatny.

Zacznijmy poznawać możliwości instrukcji `try`, modyfikując powyższy fragment kodu w taki sposób, aby zabezpieczyć go przed wystąpieniem wyjątku `FileNotFoundError`.

Przechwytywanie błędów to jeszcze nie wszystko

Gdy pojawia się błąd czasu wykonania, **zgłaszany** jest wyjątek. Jeśli *zignorujesz* zgłoszony wyjątek, będzie można go określić mianem **nieprzechwyconego**; w takiej sytuacji interpreter przerwie wykonywanie Twojego kodu, a następnie wyświetli komunikat błędu czasu wykonania (tak jak zostało to pokazane w dolnej części poprzedniej strony). Wynika stąd, że zgłoszone wyjątki mogą również zostać **przechwycone** (czyli obsłużone) za pomocą instrukcji `try`. Zauważ, że nie wystarczy jedynie przechwytywać wyjątków, trzeba *również* zdecydować, co należy zrobić potem.

Gdy zostaje zgłoszony wyjątek, może on być przechwycony lub nieprzechwycony: instrukcja try umożliwia Ci przechwycenie zgłoszonego błędu, a instrukcja except pozwala Ci coś z nim zrobić.

Być może postanowisz rozmyślnie zignorować zgłoszony wyjątek i iść dalej... mocno trzymając kciuki w intencji powodzenia. A może spróbujesz wykonać jakiś inny kod zamiast tego, który właśnie uległ awarii, a potem pójdziesz dalej. Być może też najwłaściwszą rzeczą do zrobienia okaże się zalogowanie błędu przed przerwaniem działania Twojej aplikacji w najczystszy z możliwych sposobów. Na którąkolwiek z tych opcji byś się zdecydował, zawsze pomoże Ci instrukcja `try`.

W swojej najprostszej formie instrukcja ta umożliwia Ci reagowanie w sytuacji, gdy wykonanie Twojego kodu skutkuje zgłoszeniem wyjątku. Aby chronić kod za pomocą instrukcji `try`, umieść go w jej zestawie. Gdy wystąpi wyjątek, wykonanie kodu należącego do zestawu instrukcji `try` zostanie przerwane, a następnie zostanie uruchomiony kod wchodzący w skład zestawu części `except` tej instrukcji. Zestaw części `except` to miejsce, w którym definiuje się działania, które mają być wykonane dalej.

Zaktualizujmy fragment kodu przedstawiony na poprzedniej stronie w taki sposób, aby wyświetlał krótki komunikat zawsze wtedy, gdy zgłaszany jest błąd `FileNotFoundError`. Po lewej stronie został przedstawiony kod, który miałeś wcześniej, po prawej zaś widać kod zmodyfikowany tak, aby korzystał z możliwości oferowanych przez instrukcję `try` i klauzulę `except`.

Zauważ, jak cały ten fragment kodu jest wcięty pod instrukcją try.

try_examples.py - C:/Users/lsuma...

File Edit Format Run Options Window Help

```
with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)
```

Ln: 5 Col: 0

try_examples2.py - C:\Users\lsuma\...

File Edit Format Run Options Window Help

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('Brak pliku z danymi.')
```

Ln: 8 Col: 0

Klauzula except jest wcięta tak samo, jak związana z nią instrukcja try; ma też własny zestaw kodu.

Ten kod jest wcięty pod klauzulą except; jest on wykonywany tylko wówczas, gdy pojawi się wyjątek FileNotFoundError.

Zwróć uwagę na to, że zamiast trzech wierszy kodu mamy teraz aż sześć wierszy, co może się wydawać marnotrawstwem, lecz w rzeczywistości nim nie jest. Oryginalny fragment kodu nadal jest tu obecny; tworzy on zestaw związany z instrukcją `try`. Klauzula `except` oraz związany z nią zestaw to nowy kod. Sprawdźmy, jaką różnicę robią wprowadzone tu zmiany w kodzie.

Jazda próbna

Weźmy w obroty wersję naszego kodu, w której została zastosowana konstrukcja `try...except`. Jeśli plik *myfile.txt* istnieje i może zostać odczytany przez Twój program, zawartość tego pliku pojawi się na ekranie. W przeciwnym razie zostanie zgłoszony wyjątek czasu wykonania. Wiemy już, że plik ten nie istnieje, jednak teraz zamiast oglądać paskudny komunikat śladu, który widziałeś wcześniej, możesz zobaczyć znacznie milszy dla oka komunikat z informacją, co się dzieje, który został wyświetlony przez kod obsługi wyjątku (mimo że nasz fragment kodu *w dalszym ciągu* ulega awarii). Efekt działania tego kodu został pokazany poniżej.

Za pierwszym razem, gdy uruchomiłeś fragment kodu, interpreter wygenerował ten paskudny traceback.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
========= RESTART: C:/Users/lsuma/Desktop/Python/R11/try_examples.py =========
Traceback (most recent call last):
  File "C:/Users/lsuma/Desktop/Python/R11/try_examples.py", line 2, in <module>
    with open('myfile.txt') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples2.py ========
Brak pliku z danymi.
>>>
                                                                Ln: 179  Col: 4
```

Nowa wersja kodu wyświetla znacznie przyjemniejszy komunikat dzięki zastosowaniu instrukcji try wraz z klauzulą except.

Zgłoszonych może być więcej wyjątków...

Ten nowy sposób działania naszego programu jest zdecydowanie lepszy, co się jednak stanie, gdy plik *myfile.txt* będzie istniał, lecz Twój kod nie będzie miał uprawnień do jego odczytu? Aby się o tym przekonać, utworzyliśmy odpowiedni plik i ustawiliśmy uprawnienia dostępu do niego tak, aby zasymulować tę sytuację. Ponowne uruchomienie nowego kodu skutkuje pojawieniem się na ekranie danych wyjściowych przedstawionych poniżej.

Oj! Znów widzimy paskudny komunikat tracebacku, ponieważ został zgłoszony wyjątek PermissionError.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples2.py ========
Brak pliku z danymi.
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples2.py ========
Traceback (most recent call last):
  File "C:\Users\lsuma\Desktop\Python\R11\try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
                                                                Ln: 194  Col: 4
```

Korzystaj z jednej instrukcji try, ale wielu klauzul except

Aby zabezpieczyć się przed skutkami zgłoszenia kolejnego wyjątku, do swojej instrukcji `try` po prostu dodaj kolejny zestaw związany z klauzulą `except`, wskazując wyjątek, którym jesteś zainteresowany i zapewniając dowolny kod, który wydaje Ci się niezbędny w tym nowym zestawie. Poniżej znajdziesz kolejną, zaktualizowaną wersję kodu, w którym zapewniliśmy obsługę wyjątku `PermissionError` (na wypadek gdyby został zgłoszony).

Oprócz wyjątku FileNotFoundError kod ten obsługuje też wyjątek PermissionError.

try_examples3.py - C:\Users\lsuma\Desk...

File Edit Format Run Options Window Help

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('Brak pliku z danymi.')
except PermissionError:
    print('Brak wymaganych uprawnień.')
```

Ln: 10 Col: 0

Kod umieszczony w zestawach klauzul except może robić cokolwiek. Na razie wyświetla jedynie przyjazne dla oka komunikaty.

Wykonanie tego zmodyfikowanego kodu nadal powoduje zgłoszenie wyjątku `PermissionError`. Tym razem jednak, w przeciwieństwie do poprzedniego przypadku, paskudnie wyglądający traceback został zastąpiony znacznie milszym dla oka komunikatem.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples2.py ========
Traceback (most recent call last):
  File "C:\Users\lsuma\Desktop\Python\R11\try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples3.py ========
Brak wymaganych uprawnień.
>>>
```

Ln: 198 Col: 4

To jest znacznie lepsze.

To wygląda już całkiem dobrze. Udało Ci się zmienić reakcję programu na brak pliku (tj. jego nieistnienie) lub brak dostępu do pliku (tj. brak odpowiednich uprawnień), z którego powinien korzystać. Co się jednak stanie, gdy zostanie zgłoszony wyjątek, którego się nie spodziewasz?

Mnóstwo rzeczy może pójść źle

Zanim spróbujemy odpowiedzieć na pytanie postawione na końcu ostatniej strony — *co się stanie, gdy zostanie zgłoszony wyjątek, którego się nie spodziewasz?* — przyjrzyj się niektórym wbudowanym wyjątkom języka Python 3 (które zostały skopiowane wprost z dokumentacji Pythona). Nie zdziw się, jeśli uderzy Cię to, jak wiele ich jest.

```
...
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |    +-- FloatingPointError
    |    +-- OverflowError
    |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
         +-- ModuleNotFoundError
    +-- LookupError
    |    +-- IndexError
    |    +-- KeyError
    +-- MemoryError
    +-- NameError
    |    +-- UnboundLocalError
    +-- OSError
    |    +-- BlockingIOError
    |    +-- ChildProcessError
    |    +-- ConnectionError
    |    |    +-- BrokenPipeError
    |    |    +-- ConnectionAbortedError
    |    |    +-- ConnectionRefusedError
    |    |    +-- ConnectionResetError
    |    +-- FileExistsError
    |    +-- FileNotFoundError
    |    +-- InterruptedError
    |    +-- IsADirectoryError
    |    +-- NotADirectoryError
    |    +-- PermissionError
    |    +-- ProcessLookupError
    |    +-- TimeoutError
...
```

Wszystkie wbudowane typy wyjątków dziedziczą po klasie o nazwie „Exception".

Jest ich całe mnóstwo, nie sądzisz?

Tutaj są dwa wyjątki, które nasz kod obecnie obsługuje.

Szaleństwem byłaby próba napisania osobnego zestawu klauzuli `except` dla każdego z tych wyjątków czasu wykonania, ponieważ niektóre z nich mogą się nigdy nie pojawić. Jednak inne *mogą* zostać zgłoszone, dlatego nimi powinieneś się martwić nieco bardziej. Zamiast zmuszać Cię do obsługi każdego wyjątku *z osobna*, Python umożliwia Ci zdefiniowanie zestawu **przechwytującej wszystko** klauzuli `except` wykonywanego zawsze wtedy, gdy pojawi się wyjątek czasu wykonania, który nie został przez Ciebie wyraźnie wskazany.

Kod obsługi wszystkich wyjątków

Zobaczmy, co się stanie, gdy wystąpią jakieś inne błędy. Aby zasymulować taką sytuację, plik *myfile.txt* zastąpiliśmy folderem o tej samej nazwie. Przekonajmy się, co będzie, gdy uruchomimy ten kod teraz. Uzyskany przez nas efekt możesz podziwiać poniżej.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples3.py ========
Traceback (most recent call last):
  File "C:\Users\lsuma\Desktop\Python\R11\try_examples3.py", line 3, in <module>
    with open('myfile.txt') as fh:
IsADirectoryError: (Errno 21] Is a directory: 'myfile.txt'
>>>
                                                                Ln: 9  Col: 4
```

Pojawił się kolejny wyjątek.

Zgłoszony został następny wyjątek. Mógłbyś utworzyć dodatkowy zestaw klauzuli `except`, który byłby wykonywany w odpowiedzi na wystąpienie wyjątku `IsADirectoryError`, ale zamiast tego dodajmy klauzulę przechwytującą wszystkie błędy czasu wykonania, której kod będzie uruchamiany, gdy zostanie zgłoszony *dowolny* wyjątek (poza tymi dwoma, które już wskazaliśmy). Aby to zrobić, dodaj odpowiednią instrukcję `except` na końcu istniejącego kodu.

```
try_examples4.py - C:\Users\lsuma\...
File  Edit  Format  Run  Options  Window  Help
try:
   with open('myfile.txt') as fh:
      file_data = fh.read()
   print(file_data)
except FileNotFoundError:
   print('Brak pliku z danymi.')
except PermissionError:
   print('Brak wymaganych uprawnień.')
except:
   print('Wystąpił jakiś inny błąd.')
```

Ta klauzula except jest „goła" — nie odwołuje się do żadnego określonego wyjątku.

Ten fragment kodu odpowiada za przechwycenie i obsługę wszystkich (nieprzechwyconych wcześniej) wyjątków.

Uruchomienie tej zmodyfikowanej wersji kodu pozwala się przekonać, że udało się nam pozbyć paskudnego komunikatu tracebacku i wyświetlić zamiast niego komunikat miły dla oka użytkownika. Niezależnie od tego, jakie inne wyjątki wystąpią, kod ten obsłuży je wszystkie dzięki uzupełnieniu go o przechwytującą wszystko klauzulę `except`.

To wygląda już lepiej.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples3.py ========
Traceback (most recent call last):
  File "C:\Users\lsuma\Desktop\Python\R11\try_examples3.py", line 3, in <module>
    with open('myfile.txt') as fh:
IsADirectoryError: (Errno 21] Is a directory: 'myfile.txt'
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples4.py ========
Wystąpił jakiś inny błąd.
>>>
                                                                Ln: 18  Col: 4
```

Czy czegoś właśnie nie utraciliśmy?

W porządku. Rozumiem, co się tu dzieje. Jednak czy ten kod nie ukrywa teraz faktu, że właśnie przydarzył nam się wyjątek IsADirectoryError? Czy wiedza na temat tego, z jakim dokładnie błędem mamy do czynienia, nie jest ważna?

Ach, tak... dobre pytanie.

Tworząc ostatnią wersję kodu, udało nam się doprowadzić do porządku dane wyjściowe (w tym sensie, że skutecznie pozbyliśmy się wszelkich paskudnych komunikatów tracebacku), ale utraciliśmy też przy okazji pewne istotne informacje, ponieważ nie wiemy już, jaki *konkretnie* błąd wystąpił w czasie wykonania programu.

Wiedza na temat tego, jaki wyjątek został zgłoszony, często ma dla nas znaczenie, dlatego język Python pozwala Ci dobrać się do danych związanych z ostatnim wyjątkiem *w czasie jego obsługi*. Do dyspozycji mamy tu dwa sposoby, aby to zrobić. Są nimi: użycie możliwości zapewnianych przez moduł `sys` oraz zastosowanie rozszerzenia składni instrukcji `try-except`.

Przyjrzyjmy się obydwu tym sposobom.

Nie istnieją głupie pytania

P: Czy można utworzyć kod obsługi wyłapujący wszystkie wyjątki, który nie robi absolutnie nic?

O: Tak. Bardzo kuszącą opcją wydaje się często dodanie na samym końcu instrukcji `try` zestawu klauzuli `except` o postaci przedstawionej poniżej.

```
except:
    pass
```

Prosimy, staraj się tego nie robić. Ten zestaw klauzuli `except` implementuje kod przechwytujący wszystkie błędy, który *ignoruje* wszystkie inne wyjątki (przypuszczalnie w błędnej nadziei, że gdy coś zostanie zignorowane, po prostu pójdzie sobie precz). To bardzo niebezpieczna praktyka, ponieważ wystąpienie nieoczekiwanego wyjątku powinno przynajmniej skutkować wyświetleniem na ekranie komunikatu informującego o błędzie. W związku z tym powinieneś zawsze pisać kod odporny na błędy, który obsługuje wyjątki, zamiast je po prostu ignorować.

Poznawanie wyjątków za pomocą modułu sys

Biblioteka standardowa zapewnia moduł o nazwie `sys`, który oferuje dostęp do *wnętrzności* interpretera (zestawu zmiennych i funkcji dostępnych w czasie wykonania).

Jedną z takich funkcji jest `exc_info`, która dostarcza informacji na temat obecnie obsługiwanego wyjątku. Wywołanie tej funkcji powoduje zwrócenie krotki zawierającej trzy wartości. Pierwsza z nich wskazuje **typ** wyjątku, druga zawiera szczegóły dotyczące jego **wartości**, a trzecia przechowuje **obiekt śladu** zapewniający dostęp do komunikatu tracebacku (gdybyś z jakichś powodów go potrzebował). Jeśli nie ma żadnego dostępnego wyjątku, funkcja `exc_info` zwraca pythonową wartość null w przypadku każdej wartości należącej do krotki, co wygląda tak: `(None, None, None)`.

Aby dowiedzieć się więcej na temat modułu sys, odwiedź stronę *https://docs.python.org/3/library/sys.html.*

Wiedząc to wszystko, poeksperymentujmy trochę w wierszu poleceń środowiska IDLE. W ramach przedstawionej poniżej sesji napisaliśmy trochę kodu, który zawsze doprowadzi do błędu czasu wykonania (ponieważ dzielenie przez zero *nigdy* nie jest dobrym pomysłem). W zestawie klauzuli `except` wyłapującej wszystkie wyjątki wykorzystaliśmy funkcję `sys.exc_info`, aby wydobyć i wyświetlić dane związane z bieżącym wyjątkiem.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
================================ RESTART: Shell ================================
>>>
>>> import sys
>>>
>>> try:
        1/0
except:
        err = sys.exc_info()
        for e in err:
                print(e)

<class 'ZeroDivisionError'>
division by zero
<traceback object at 0x0000027D57D9AE48>
>>> |
                                                                Ln: 19  Col: 4
```

Pamiętaj o zaimportowaniu modułu sys.

Dzielenie przez zero NIGDY nie jest dobrym pomysłem... a gdy wykonasz taką operację w swoim kodzie, pojawi się wyjątek.

Wydobądźmy i wyświetlmy dane związane z bieżącym wyjątkiem.

Tutaj widać dane związane z naszym wyjątkiem, które potwierdzają, że mamy do czynienia z problemem dzielenia przez zero.

Można wgłębić się bardziej w obiekt tracebacku, aby dowiedzieć się więcej o tym, co się właśnie stało, ale wydaje się to zbyt pracochłonne, prawda? Wszystkim, czego nam tak naprawdę potrzeba, jest informacja na temat *typu* zgłoszonego wyjątku.

Aby uprościć to zadanie (a zarazem ułatwić Ci życie), język Python rozszerza składnię instrukcji `try-except`, sprawiając, że dostęp do informacji zwracanych przez funkcję `sys.exc_info` realizowany jest w bardziej wygodny sposób, a więc bez zmuszania Cię do pamiętania o importowaniu modułu `sys` i użerania się z krotką zwracaną przez tę funkcję.

Przypomnij sobie informację przedstawioną kilka stron temu. Zgodnie z nią wyjątki mają pewną hierarchię, w której każdy z nich dziedziczy po jednym wyjątku, noszącym nazwę `Exception`. Skorzystajmy z istnienia tej hierarchii, aby napisać na nowo nasz kod obsługi wszystkich wyjątków.

Przypomnij sobie hierarchię wyjątków przedstawioną wcześniej w tym rozdziale.

```
...
+-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |     +-- FloatingPointError
      |     +-- OverflowError
      |     +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
...
```

Zrewidowany kod obsługi wszystkich wyjątków

Przyjrzyj się swojemu (przedstawionemu poniżej) bieżącemu kodowi, w którym jawnie wskazałeś dwa rodzaje wyjątków, jakie postanowiłeś obsłużyć (błędy `FileNotFoundError` oraz `PermissionError`), jak również zapewniłeś zestaw związany z ogólną, przechwytującą wszystko klauzulą `except`.

Ten kod działa, ale tak naprawdę nie daje nam wiele informacji o nieoczekiwanym wyjątku.

try_examples4.py - C:\Users\lsuma\...

File Edit Format Run Options Window Help

```
try:
   with open('myfile.txt') as fh:
      file_data = fh.read()
   print(file_data)
except FileNotFoundError:
   print('Brak pliku z danymi.')
except PermissionError:
   print('Brak wymaganych uprawnień.')
except:
   print('Wystąpił jakiś inny błąd.')
```

Ln: 12 Col: 0

Zauważ, że odwołując się do *konkretnego* wyjątku, wskazywaliśmy go za pomocą nazwy po słowie kluczowym `except`. W podobny sposób można wskazywać całe *klasy* wyjątków, używając dowolnych nazw występujących w hierarchii.

Przypomnij sobie hierarchię wyjątków przedstawioną wcześniej w tym rozdziale.

```
...
+-- Exception
     +-- StopIteration
     +-- StopAsyncIteration
     +-- ArithmeticError
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
...
```

Jeśli jesteś na przykład zainteresowany tym, czy został zgłoszony jakiś wyjątek arytmetyczny (a nie — konkretnie — tym, że wystąpił błąd dzielenia przez zero), możesz skorzystać z klauzuli `except ArithmeticError`, dzięki której przechwycisz błędy `FloatingPointError`, `OverflowError` oraz `ZeroDivisionError`, gdy tylko wystąpią. Podobnie, jeśli użyjesz klauzuli `except Exception`, będziesz przechwytywał *dowolne* błędy.

W jaki jednak sposób może Ci to pomóc, zwłaszcza że przechwytujesz już wszystkie błędy za pomocą „gołej" instrukcji `except`? To prawda, przechwytujesz. Możesz jednak rozszerzyć klauzulę `except Exception` za pomocą słowa kluczowego `as`, pozwalającego Ci przypisać obiekt bieżącego wyjątku do zmiennej (często stosuje się w tej roli zmienną o nazwie `err`) i utworzyć bardziej pouczający komunikat błędu. Przyjrzyj się kolejnej, przedstawionej poniżej wersji kodu, w której została zastosowana konstrukcja `except Exception as`.

try_examples5.py - C:\Users\lsuma\Desktop\Python\...

File Edit Format Run Options Window Help

```
try:
   with open('myfile.txt') as fh:
      file_data = fh.read()
   print(file_data)
except FileNotFoundError:
   print('Brak pliku z danymi.')
except PermissionError:
   print('Brak wymaganych uprawnień.')
except Exception as err:
   print('Wystąpił jakiś inny błąd:', str(err))
```

Ln: 12 Col: 0

W przeciwieństwie do „gołej", przechwytującej wszystkie wyjątki instrukcji except, która została pokazana powyżej, ta sprawia, że obiekt wyjątku zostaje przypisany do zmiennej err.

Wartość zmiennej err jest następnie wykorzystana jako część komunikatu przyjaznego użytkownikowi (jako że dobrym pomysłem jest raportowanie wszystkich pojawiających się wyjątków).

Jazda próbna

Wprowadziwszy tę — już ostatnią — zmianę w kodzie prezentującym możliwości instrukcji `try-except`, potwierdźmy, że wszystko działa zgodnie z naszymi oczekiwaniami, zanim powrócimy do aplikacji WWW *vsearch4web.py* i zastosujemy w kodzie modyfikacje związane z obsługą wyjątków.

Zacznijmy od sprawdzenia, czy nasz kod wyświetla odpowiedni komunikat, gdy nie ma wymaganego pliku.

```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Brak pliku z danymi.
>>>
Ln: 20 Col: 4
```

Plik myfile.txt nie istnieje.

Jeśli odpowiedni plik istnieje, lecz nie masz uprawnień, aby uzyskać do niego dostęp, zgłoszony zostaje inny wyjątek.

```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Brak pliku z danymi.
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Brak wymaganych uprawnień.
>>>
Ln: 29 Col: 4
```

Plik istnieje, lecz nie można go odczytać.

Wszelkie inne wyjątki zostają obsłużone za pomocą wychwytującej wszystko klauzuli `except`, której zestaw wyświetla przyjazny komunikat z informacją o błędzie.

```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Brak pliku z danymi.
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Brak wymaganych uprawnień.
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Wystąpił jakiś inny błąd: [Errno 21] Is a directory: 'myfile.txt'
>>>
Ln: 32 Col: 4
```

Wystąpił jakiś inny wyjątek. W tym przypadku coś, co uważałeś za plik, w rzeczywistości okazało się folderem.

Wreszcie, gdy wszystko jest w porządku, zestaw instrukcji `try` zostaje wykonany bez błędów, a zawartość pliku jest wyświetlana na ekranie.

```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Brak pliku z danymi.
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Brak wymaganych uprawnień.
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Wystąpił jakiś inny błąd: [Errno 21] Is a directory: 'myfile.txt'
>>>
======== RESTART: C:\Users\lsuma\Desktop\Python\R11\try_examples5.py ========
Pusty (z wyjątkiem tego jednego wiersza).
>>>
Ln: 35 Col: 4
```

Sukces! Nie wystąpiły żadne wyjątki, dzięki czemu zestaw instrukcji try został wykonany w całości.

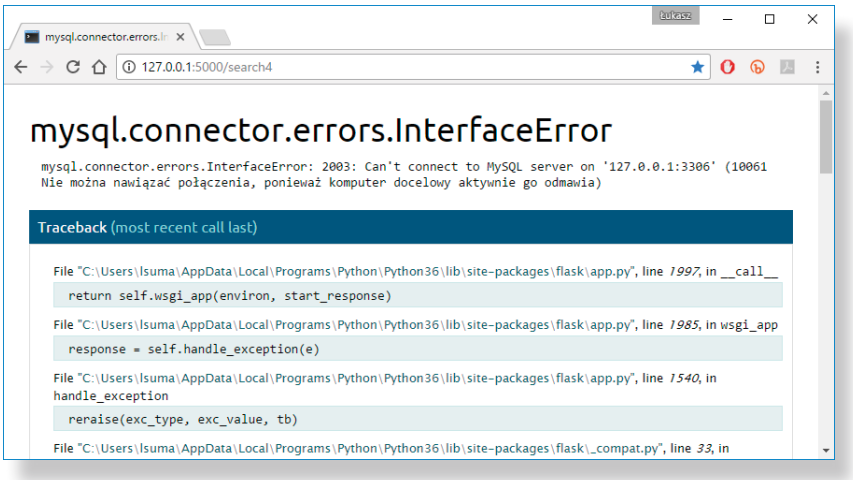
Powrót do kodu naszej aplikacji WWW

Przypomnij sobie z początku tego rozdziału, że wskazaliśmy pewien problem związany z wywołaniem funkcji `log_request` z wnętrza funkcji `do_search` należącej do kodu programu *vsearch4web.py*. Konkretnie martwiło nas tam, co należy zrobić, gdy wywołanie to się nie powiedzie.

```
...
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
...
```

4. Co się stanie, gdy ta operacja się nie uda?

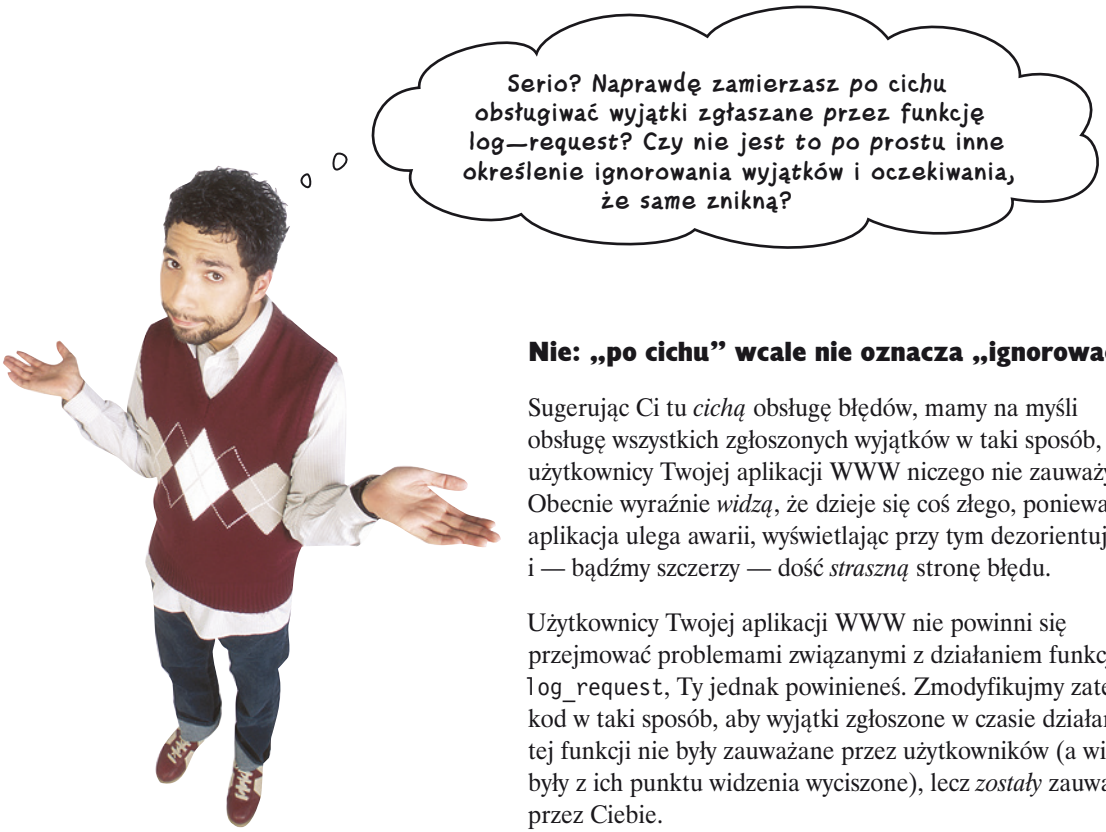
Dzięki naszym dociekaniom wiemy już, że wywołanie to może się nie powieść, jeśli baza danych pracująca po stronie backendu będzie niedostępna lub jeśli wystąpi jakiś inny błąd. Jeśli pojawi się błąd (dowolnego typu), aplikacja WWW zareaguje, wyświetlając dość nieprzyjaźnie wyglądającą stronę, która prawdopodobnie raczej zdezorientuje użytkowników, zamiast poinformować ich o problemie w odpowiedni sposób.



To nie jest coś, co chciałbyś pokazywać użytkownikom swojej aplikacji WWW.

Chociaż jest to ważne z naszego punktu widzenia, logowanie każdego żądania sieciowego obsługiwanego przez aplikację WWW nie jest czymś, co szczególnie interesuje jej użytkowników; wszystkim, co chcą oni widzieć, są wyniki ich wyszukiwań. W związku z tym zmieńmy kod naszej aplikacji w taki sposób, aby radziła ona sobie z błędami pojawiającymi się przy okazji wykonywania funkcji `log_request`, obsługując wszystkie zgłoszone wyjątki *po cichu*.

Cicha obsługa wyjątków



Nie: „po cichu" wcale nie oznacza „ignorować".

Sugerując Ci tu *cichą* obsługę błędów, mamy na myśli obsługę wszystkich zgłoszonych wyjątków w taki sposób, aby użytkownicy Twojej aplikacji WWW niczego nie zauważyli. Obecnie wyraźnie *widzą*, że dzieje się coś złego, ponieważ aplikacja ulega awarii, wyświetlając przy tym dezorientującą i — bądźmy szczerzy — dość *straszną* stronę błędu.

Użytkownicy Twojej aplikacji WWW nie powinni się przejmować problemami związanymi z działaniem funkcji `log_request`, Ty jednak powinieneś. Zmodyfikujmy zatem kod w taki sposób, aby wyjątki zgłoszone w czasie działania tej funkcji nie były zauważane przez użytkowników (a więc były z ich punktu widzenia wyciszone), lecz *zostały* zauważone przez Ciebie.

Nie istnieją głupie pytania

P: Czy wszystkie te instrukcje `try-except` nie sprawiają tylko, że mój kod staje się mniej czytelny i trudniejszy do zrozumienia?

O: To prawda, że przykładowy kod, z którego korzystamy w tym rozdziale, składał się na samym początku z zaledwie trzech wierszy, aby z czasem rozrosnąć się o kolejne siedem, które — z pozoru — nie mają nic wspólnego z tym, co robią te trzy oryginalne. Ważne jest jednak, aby chronić kod, który może potencjalnie zgłosić wyjątek, a zastosowanie konstrukcji `try-except` jest generalnie uważane za najlepszy sposób, aby to zrobić. Z czasem nauczysz się dostrzegać istotne rzeczy (czyli kod odpowiedzialny za właściwe działanie) w obrębie zestawu instrukcji `try` i odfiltrowywać zestawy związane z klauzulami `except`, których jedynym zadaniem jest obsługa wyjątków. Próbując zrozumieć kod, w którym używane są konstrukcje `try-except`, najpierw czytaj zawsze zestaw instrukcji `try`, aby dowiedzieć się, co robi dany kod, a następnie dopiero — jeśli musisz rozumieć, co się dzieje, gdy sprawy idą źle — spoglądaj w zestawy związane z klauzulami `except`.

Zaostrz ołówek

Dołóżmy trochę kodu wykorzystującego konstrukcję `try-except` do wywołania funkcji `log_request` znajdującego się wewnątrz funkcji `do_search`. Aby uprościć sprawę, wokół wywołania funkcji `log_request` dodajmy przechwytujący wszystkie wyjątki kod obsługi, za pomocą którego na standardowym wyjściu będziemy wyświetlali przydatny komunikat (korzystając w tym celu z wbudowanej funkcji `print`). Definiując ten kod w taki sposób, możesz wyłączyć standardowe zachowanie Twojej aplikacji związane z obsługą wyjątków, które polega obecnie na wyświetlaniu nieprzyjaznej strony błędu.

Poniżej znajdziesz kod funkcji `log_request` w takiej postaci, jaką ma on w tej chwili.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Ten wiersz kodu powinien być chroniony na wypadek awarii (zgłoszenia błędu czasu wykonania).

W pustych miejscach poniżej, wokół wywołania funkcji `log_request`, wpisz kod zapewniający obsługę wszystkich wyjątków.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))

      ..................................

              ..............................................................

      ..............................................................

              ..............................................................

    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Nie zapomnij o wywołaniu funkcji log_request stanowiącym część składową kodu, który powinieneś dodać.

Zaostrz ołówek
Rozwiązanie

Nasz plan polegał na dołożeniu kodu wykorzystującego konstrukcję `try-except` do wywołania funkcji `log_request` znajdującego się wewnątrz funkcji `do_search`. Aby uprościć sprawę, wokół wywołania funkcji `log_request` postanowiliśmy dodać przechwytujący wszystkie wyjątki kod obsługi, za pomocą którego na standardowym wyjściu chcieliśmy wyświetlać przydatny komunikat (korzystając w tym celu z wbudowanej funkcji `print`).

Poniżej znajdziesz kod funkcji `log_request` w takiej postaci, jaką ma on w tej chwili.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

W pustych miejscach poniżej, wokół wywołania funkcji `log_request`, miałeś wpisać kod zapewniający obsługę wszystkich wyjątków.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logowanie się nie powiodło; wystąpił błąd:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

To wywołanie funkcji log_request zostało przeniesione do wnętrza zestawu związanego z nową instrukcją try.

Ta klauzula przechwytuje wszystkie wyjątki.

Gdy wystąpi błąd czasu wykonania, ten komunikat zostanie wyświetlony jedynie na ekranie administratora. Użytkownicy aplikacji nie będą niczego widzieli.

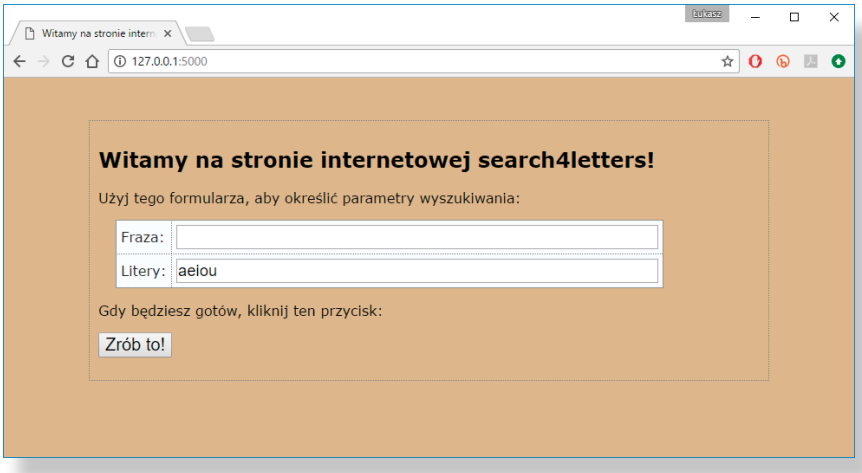
(Rozszerzona) jazda próbna (1. z 3)

Skoro dodaliśmy już do aplikacji *vsearch4web.py* przechwytujący wszystko kod obsługi wyjątków, weźmy ją w bardziej gruntowne obroty (obejmujące kilka kolejnych stron), aby się przekonać, jaką różnicę robią wprowadzone przez nas zmiany. Wcześniej, gdy coś poszło nie tak, użytkownik oglądał niezbyt ciekawie wyglądającą stronę błędu. Teraz jednak błąd ten jest obsługiwany „po cichu" dzięki zastosowaniu wyłapującego wszystko kodu obsługi. Jeśli nie zrobiłeś tego do tej pory, uruchom program *vsearch4web.py*, a następnie skorzystaj z dowolnej przeglądarki internetowej, aby otworzyć stronę domową naszej aplikacji WWW, tak jak zostało to pokazane poniżej.

```
C:\webapp>py -3 vsearch4web.py
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 780-870-098
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Twoja aplikacja została uruchomiona i działa, oczekując poleceń nadchodzących z przeglądarki...

Śmiało, przejdź do strony domowej Twojej aplikacji WWW.



W oknie terminala, w którym uruchomiłeś swoją aplikację, powinieneś widzieć komunikaty podobne do przedstawionych poniżej.

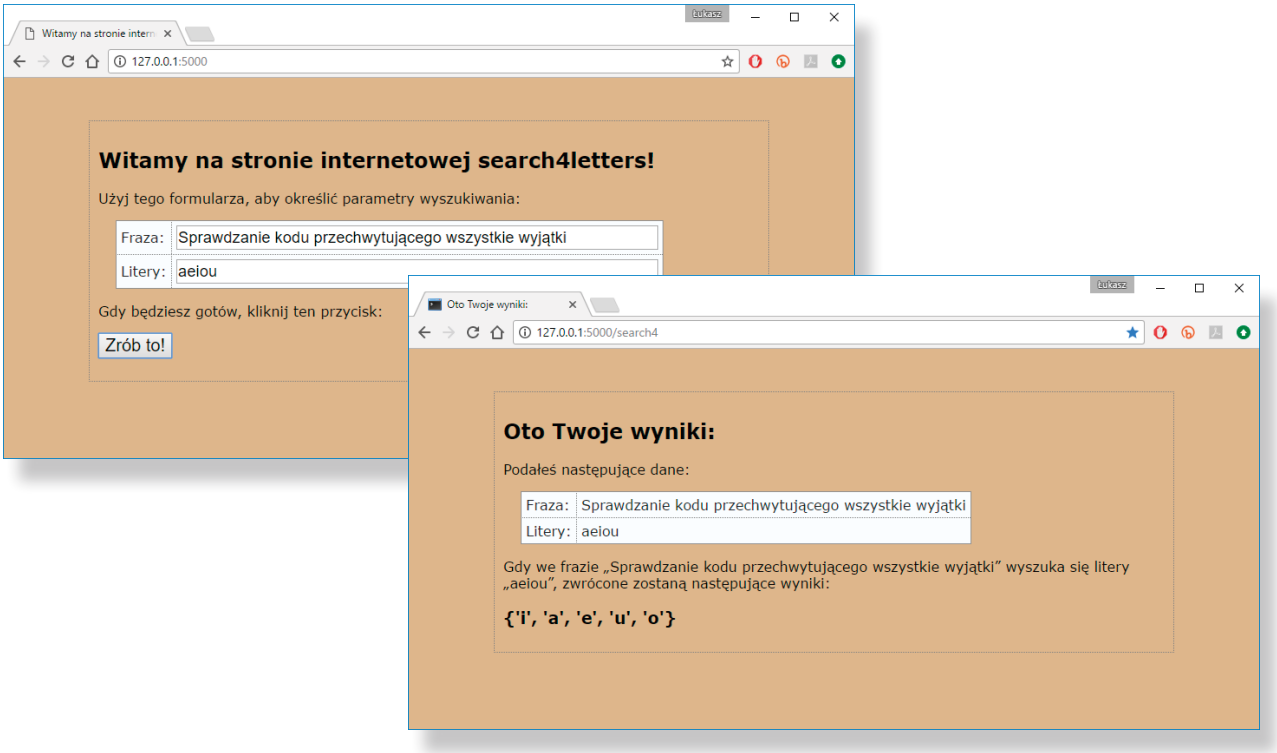
```
...
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [01/Jul/2017 18:53:58] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [01/Jul/2017 18:53:58] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [01/Jul/2017 18:53:59] "GET /favicon.ico HTTP/1.1" 404 -
```

Te kody stanu (200) potwierdzają, że Twoja aplikacja WWW została uruchomiona i działa (a także wyświetliła swoją stronę domową). Jak do tej pory wszystko jest w porządku.

Tak przy okazji: nie martw się tym kodem błędu (404). Nie zdefiniowaliśmy w naszej aplikacji pliku favicon z grafiką (dlatego nieodnalezienie tego pliku zostało zgłoszone, gdy Twoja przeglądarka poprosiła o jego dostarczenie).

(Rozszerzona) jazda próbna (2. z 3)

Aby zasymulować wystąpienie błędu, wyłączyliśmy bazę danych pracującą po stronie backendu, co powinno spowodować pojawienie się błędu za każdym razem, gdy aplikacja WWW spróbuje skomunikować się z bazą. Ponieważ Twój kod po cichu przechwytuje wszystkie błędy spowodowane przez funkcję `log_request`, użytkownik aplikacji nie zostaje poinformowany o tym, że logowanie nie doszło do skutku. Kod ten generuje również wyświetlany na ekranie komunikat opisujący problem. Oczywiście, gdy wprowadzisz frazę i naciśniesz przycisk *Zrób to!*, aplikacja WWW wyświetli wyniki wyszukiwania w oknie przeglądarki internetowej, a w oknie terminala zostanie pokazany „wyciszony" komunikat błędu. Zauważ, że pomimo wystąpienia błędu czasu wykonania aplikacja kontynuuje działanie i z powodzeniem obsługuje wywołanie URL-a `/search`, co pokazano poniżej.




```
...
127.0.0.1 - - [01/Jul/2017 18:53:59] "GET /favicon.ico HTTP/1.1" 404 -
***** Logowanie się nie powiodło; wystąpił błąd: 2003: Can't connect to MySQL server on '127.0.0.1:3306'
(10061 Nie można nawiązać połączenia, ponieważ komputer docelowy aktywnie go odmawia)
127.0.0.1 - - [01/Jul/2017 19:28:51] "POST /search4 HTTP/1.1" 200 -
```

Ten komunikat jest generowany przez Twój kod obsługi wychwytujący wszystkie wyjątki. Użytkownik aplikacji WWW go nie widzi.

Mimo że wystąpił błąd, aplikacja nie uległa awarii. Innymi słowy, wyszukiwanie zostało przeprowadzone (lecz użytkownik aplikacji nie ma świadomości, że logowanie się nie powiodło).

(Rozszerzona) jazda próbna (3. z 3)

W gruncie rzeczy nieważne jest, jaki błąd wystąpi w czasie działania funkcji `log_request`, ponieważ nasz przechwytujący wszystko kod obsługi zajmie się każdym błędem.

Zrestartowaliśmy naszą bazę danych działającą po stronie backendu, a następnie spróbowaliśmy się z nią połączyć, korzystając z niewłaściwej nazwy użytkownika. Wystąpienie tego błędu możesz sprowokować, zmieniając zawartość słownika `dbconfig` w kodzie programu *vsearch4web.py* w taki sposób, aby wartością powiązaną z kluczem `user` był łańcuch `vsearchwrong`.

```
...
app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearchwrong',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }
...
```

Gdy Twoja aplikacja WWW załaduje się ponownie, a Ty uruchomisz wyszukiwanie, w oknie terminala ujrzysz komunikat podobny do przedstawionego niżej.

```
...
***** Logowanie się nie powiodło; wystąpił błąd: 1045 (28000): Access denied for user 'vsearchwrong'@'localhost'
 (using password: YES)
```

Ponownie ustaw nazwę użytkownika na `vsearch`, a następnie spróbuj odwołać się do nieistniejącej tabeli, zmieniając nazwę tabeli w zapytaniu SQL używanym w funkcji `log_request` na `logwrong` (zamiast poprawnej `log`).

```
def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into logwrong
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
...
```

Gdy aplikacja załaduje się ponownie, a Ty przeprowadzisz wyszukiwanie, w oknie terminala zobaczysz komunikat podobny do widocznego poniżej.

```
...
***** Logowanie się nie powiodło; wystąpił błąd: 1146 (42S02): Table 'vsearchlogdb.logwrong' doesn't exist
```

Zmień nazwę tabeli z powrotem na `log`, a następnie wykonaj ostatni eksperyment. Dodaj do funkcji `log_request` generującą własny wyjątek instrukcję `raise` (w ten sam sposób, jak wcześniej zrobiłeś to z instrukcją `with`).

```
def log_request(req: 'flask_request', res: str) -> None:
    raise Exception("Właśnie stało się coś okropnego.")
    with UseDatabase(app.config['dbconfig']) as cursor:
...
```

Gdy Twoja aplikacja WWW załaduje się ponownie już po raz ostatni, a Ty uruchomisz wyszukiwanie, w oknie terminala ujrzysz poniższy komunikat.

```
...
***** Logowanie się nie powiodło; wystąpił błąd: Właśnie stało się coś okropnego.
```

Obsługa innych błędów bazy danych

Funkcja `log_request` wykorzystuje menedżer kontekstu `UseDatabase` (zapewniany przez moduł `DBcm`). Teraz, gdy zabezpieczyłeś już wywołanie tej funkcji, możesz się zrelaksować, mając świadomość, że wszelkie problemy związane z bazą danych zostaną przechwycone (i obsłużone) przez Twój przechwytujący wszystko kod obsługi wyjątków.

Jednak funkcja `log_request` to niejedyne miejsce, w którym Twoja aplikacja komunikuje się z bazą danych. Innym jest funkcja `view_the_log`, która pobiera zalogowane dane z bazy, a następnie wyświetla je na ekranie.

Przypomnij sobie kod funkcji `view_the_log`, który został zaprezentowany poniżej.

Cały ten kod również powinien być chroniony.

```
...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
    ...
```

Również ten kod może zawieść, jako że także on komunikuje się z bazą danych pracującą po stronie backendu. Jednak w przeciwieństwie do `log_request` funkcja `view_the_log` nie jest wywoływana z poziomu kodu aplikacji *vsearch4web.py*, lecz w Twoim imieniu wywołuje ją framework Flask. Oznacza to, że nie możesz napisać kodu chroniącego to wywołanie, ponieważ odbywa się ono w ramach frameworka, nie Twojego kodu.

Skoro nie możesz zabezpieczyć wywołania funkcji `view_the_log`, następnym dobrym rozwiązaniem będzie ochrona kodu wewnątrz jej zestawu, a szczególnie fragmentu, w którym następuje użycie menedżera kontekstu `UseDatabase`. Zanim zajmiemy się rozważeniem sposobu zrobienia tego, zastanówmy się, co może tu pójść nie tak:

- Pracująca po stronie backendu baza danych może być niedostępna.
- Możesz nie być w stanie zalogować się do działającej bazy danych.
- Po udanym zalogowaniu się może zawieść wykonanie zapytania bazodanowego.
- Może się zdarzyć coś innego (niespodziewanego).

Przedstawiona powyżej lista potencjalnych problemów przypomina spis rzeczy, którymi powinieneś był się martwić w przypadku funkcji `log_request`.

Czy „więcej błędów" oznacza „więcej klauzul except"?

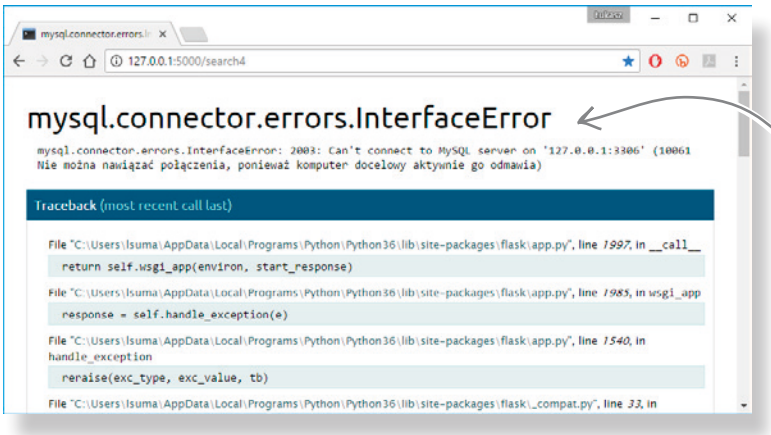
Dysponując już pewną wiedzą na temat konstrukcji `try-except`, moglibyśmy dodać do funkcji `view_the_log` trochę kodu w celu zabezpieczenia użycia menedżera kontekstu `UseDatabase`.

```
...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
        ...
    except Exception as err:
        print('Coś poszło źle:', str(err))
```

Pozostała część kodu funkcji powinna się znaleźć tutaj.

Kolejny kod obsługi wyjątków, który przechwytuje wszystko.

Ta strategia przechwytywania wszystkiego z pewnością działa (w końcu to właśnie z niej korzystałeś w przypadku funkcji `log_request`). Sprawy się jednak skomplikują, gdy zdecydujesz się zrobić coś innego niż implementować kod obsługi wyjątków, który przechwytuje wszystko. Co będzie, gdy postanowisz reagować na jakiś konkretny błąd bazy danych, taki jak ten, który pojawia się, gdy baza nie zostaje odnaleziona? Przypomnij sobie z początku tego rozdziału, że baza MySQL zgłasza wyjątek `InterfaceError`, gdy dzieje się coś takiego, jak pokazano poniżej.



Ten wyjątek zostaje zgłoszony, gdy Twój kod nie może odnaleźć bazy danych pracującej po stronie backendu.

Mógłbyś tu dodać klauzulę `except` związaną z wyjątkiem `InterfaceError`, jednak aby dało się to zrobić, musiałbyś również zaimportować w swoim kodzie moduł `mysql.connector`, w którym został zdefiniowany ten konkretny wyjątek.

Z pozoru nie wydaje się to wielkim problemem. W rzeczywistości jednak nim jest.

Unikaj ściśle sprzężonego kodu

Załóżmy, że postanowiłeś utworzyć klauzulę `except`, która zabezpiecza Twój program przed wystąpieniem błędu związanego z brakiem dostępu do bazy danych. Mógłbyś w takiej sytuacji zmienić kod funkcji `view_the_log` tak, aby wyglądał jak poniżej.

```
    ...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
   try:
      with UseDatabase(app.config['dbconfig']) as cursor:
            ...

   except mysql.connector.errors.InterfaceError as err:
      print('Czy Twoja baza danych jest włączona? Błąd:', str(err))
   except Exception as err:
      print('Coś poszło źle:', str(err))
            ...
```

Pozostała część kodu funkcji nadal powinna być tutaj.

Dodaj kolejną klauzulę except, aby obsłużyć konkretny wyjątek.

Jeśli będziesz tu pamiętał o dodaniu instrukcji `import mysql.connector` na początku swojego kodu, ta dodatkowa klauzula `except` zadziała. Gdy Twojej bazy danych działającej po stronie backendu nie uda się odnaleźć, ten dodatkowy kod umożliwi Twojej aplikacji WWW przypomnienie Ci, że powinieneś sprawdzić, czy baza ta jest uruchomiona.

Nowo dodany kod działa, a Ty widzisz, co się tutaj dzieje, więc... o co chodzi?

Z przedstawionym tu rozwiązaniem wiąże się problem polegający na tym, że kod programu *vsearch4web.py* jest obecnie bardzo **ściśle sprzężony** z bazą danych *MySQL*, a konkretnie — z zastosowaniem modułu *MySQL Connector*. Przed dodaniem tej drugiej klauzuli `except` kod Twojego programu komunikował się z bazą danych wyłącznie za pośrednictwem modułu `DBcm` (który opracowaliśmy we wcześniejszej części tej książki). W szczególności menedżer kontekstu `UseDatabase` zapewniał wygodną warstwę **abstrakcji** (czy też po prostu separację), która oddzielała kod znajdujący się w pliku *vsearch4web.py* od bazy danych działającej po stronie backendu. Gdybyś za jakiś czas postanowił z jakichś względów zastąpić bazę *MySQL* bazą *PostgreSQL*, zmiany musiałbyś wprowadzić jedynie w module `DBcm`, *nie* zaś w całym kodzie wykorzystującym klasę `UseDatabase`. Gdy jednak tworzysz kod podobny do przedstawionego powyżej, ściśle wiążesz (czyli sprzęgasz) kod swojej aplikacji WWW z bazą danych MySQL przez użycie instrukcji `import mysql.connector`, jak również dodanie nowej klauzuli `except` odwołującej się do wyjątku `mysql.connector.errors.InterfaceError`.

Gdy musisz napisać kod, który wprowadza ścisłe sprzężenie z Twoją bazą danych, zawsze rozważ umieszczenie go w module `DBcm`. Dzięki temu Twoja aplikacja WWW zostanie opracowana w taki sposób, aby wykorzystywać ogólny interfejs zapewniany przez ten moduł zamiast specyficznego interfejsu komunikującego się (i wiążącego Cię) z konkretną bazą danych.

Zastanówmy się teraz, co dla naszej aplikacji WWW oznacza przeniesienie przedstawionego powyżej kodu związanego z klauzulą `except` do modułu `DBcm`.

Zrewidowany moduł DBcm

Do kodu modułu DBcm zaglądaliśmy po raz ostatni w rozdziale 9., gdy go tworzyliśmy, aby umożliwić sobie wykorzystywanie instrukcji with do działań związanych z bazą danych *MySQL*. Wówczas odłożyliśmy na bok wszelkie kwestie związane z obsługą błędów (wygodnie ignorując cały ten problem). Teraz, gdy miałeś okazję się przekonać, co oferuje Ci funkcja sys.exc_info, powinieneś mieć już nieco większe pojęcie o tym, co oznaczają argumenty metody __exit__ należącej do klasy UseDatabase. Raz jeszcze rzuć okiem na kod modułu DBcm.

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

To jest kod menedżera kontekstu znajdujący się w pliku DBcm.py.

Teraz, gdy widziałeś już funkcję exc_info, powinno być dla Ciebie jasne, do czego odnoszą się te argumenty; chodzi tu o dane wyjątku.

Przypomnij sobie, że klasa UseDatabase implementuje trzy metody:

- metodę __init__, która zapewnia okazję do konfiguracji *przed* wykonaniem instrukcji with;
- metodę __enter__, która zostaje wykonana, gdy instrukcja with *zaczyna* działać;
- metodę __exit__, która na pewno zostanie uruchomiona, gdy zestaw instrukcji with się *zakończy*.

Tak przynajmniej wygląda spodziewany sposób działania, gdy wszystko idzie zgodnie z planem. Gdy coś idzie nie tak, zachowanie to ulega **zmianie**.

Na przykład gdy podczas wykonywania metody __enter__ zostanie zgłoszony wyjątek, instrukcja with jest przerywana, a wszelkie dalsze przetwarzanie metody __exit__ zostaje *anulowane*. Ma to sens: jeśli metoda __enter__ „wpadnie w kłopoty”, w przypadku metody __exit__ nie można już zakładać, że kontekst wykonania został właściwie zainicjalizowany i skonfigurowany (dlatego rozsądniej jest w ogóle nie uruchamiać kodu metody __exit__).

Głównym problemem związanym z kodem metody __enter__ jest to, że używana w niej baza danych może nie być dostępna, dlatego poświęćmy trochę czasu na zmodyfikowanie tej metody w taki sposób, aby przygotować ją na tę ewentualność, generując własny wyjątek, gdy nie da się nawiązać połączenia z bazą. Po zrobieniu tego zmienimy funkcję view_the_log tak, aby przechwytywała ten nasz wyjątek zamiast wyjątku mysql.connector.errors.InterfaceError, bardzo specyficznego dla zastosowanej bazy danych.

Tworzenie własnego wyjątku

Tworzenie własnego (czyli niestandardowego) wyjątku nie mogłoby być prostsze. Wystarczy tu wybrać odpowiednią nazwę, a następnie zdefiniować pustą klasę dziedziczącą po wbudowanej klasie `Exception` języka Python. Po zrobieniu tego swój wyjątek będziesz mógł zgłosić, korzystając ze słowa kluczowego `raise`. A gdy to się stanie, będziesz mógł go przechwycić (i odpowiednio obsłużyć) za pomocą konstrukcji `try-except`.

Szybka wycieczka do wiersza poleceń środowiska IDLE pomoże nam zaprezentować własne wyjątki w akcji. W przedstawionym poniżej przykładzie został utworzony własny wyjątek o nazwie `ConnectionError`, który został następnie zgłoszony (za pomocą instrukcji `raise`), potem zaś przechwycony przy użyciu konstrukcji `try-except`. Przeczytaj zamieszczone poniżej komentarze w kolejności wskazywanej przez ich numery i (jeśli chcesz być na bieżąco) wprowadź w swoim wierszu poleceń kod, z którego korzystaliśmy.

1. Utwórz nową klasę o nazwie „ConnectionError", która dziedziczy po klasie Exception.

2. W Pythonie pass to instrukcja pusta, dzięki której można utworzyć pustą klasę.

3. Zgłoszenie naszego nowego wyjątku (za pomocą instrukcji raise) powoduje wyświetlenie komunikatu śladu.

4. Przechwyć wyjątek ConnectionError za pomocą konstrukcji try-except.

5. Wyjątek ConnectionError został przechwycony, umożliwiając nam dostosowanie komunikatu błędu.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>>
>>> class ConnectionError(Exception):
        pass

>>>
>>>
>>> raise ConnectionError('Nie mogę się połączyć... Czy to już czas na panikę?')
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    raise ConnectionError('Nie mogę się połączyć... Czy to już czas na panikę?')
ConnectionError: Nie mogę się połączyć... Czy to już czas na panikę?
>>>
>>>
>>> try:
        raise ConnectionError('Ups!')
except ConnectionError as err:
        print('Mam:', str(err))

Mam: Ups!
>>>
>>>
                                                                    Ln: 79  Col: 4
```

Ta pusta klasa nie jest aż tak pusta...

Pisząc, że klasa `ConnectionError` jest pusta, troszkę skłamaliśmy. Oczywiście zastosowanie słowa kluczowego `pass` spowodowało, że nie znalazł się w niej żaden *nowy* kod, jednak fakt, że **dziedziczy** ona po wbudowanej klasie `Exception` języka Python, oznacza, że wszystkie atrybuty i zachowania tej ostatniej są również dostępne w klasie `ConnectionError` (co sprawia, że zdecydowanie nie jest ona pusta). Wyjaśnia to, dlaczego klasa ta działa dokładnie tak, jak tego oczekujesz, gdy posługujesz się nią wraz z instrukcjami `raise` oraz `try-except`.

Zaostrz ołówek

1. Poprawmy moduł DBcm w taki sposób, aby zgłaszał niestandardowy wyjątek `ConnectionError` zawsze wtedy, gdy połączenie z bazą danych działającą po stronie backendu się nie powiedzie.

   Poniżej znajdziesz aktualny kod modułu `DBcm.py`. W pustych miejscach dodaj kod niezbędny do zgłoszenia wyjątku `ConnectionError`.

Tutaj zdefiniuj własny wyjątek.

```
import mysql.connector

.....................................................................................

    ...........................

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':

        ...........................

        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

        .....................................................................................

                ...........................................................................

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Dodaj kod zgłaszający wyjątek ConnectionError.

2. Zmieniwszy kod modułu DBcm, użyj ołówka, aby opisać wszystkie modyfikacje, które wprowadziłbyś w przedstawionym poniżej w skrócie kodzie programu *vsearch4web.py*, żeby móc skorzystać z nowo zdefiniowanego wyjątku `ConnectionError`.

Użyj ołówka, aby wskazać zmiany, które należałoby wprowadzić w tym kodzie teraz, gdy istnieje już wyjątek ConnectionError.

```
from DBcm import UseDatabase
import mysql.connector
...
                                the_row_titles=titles,
                                the_data=contents,)
    except mysql.connector.errors.InterfaceError as err:
        print('Czy Twoja baza danych jest włączona? Błąd:', str(err))
    except Exception as err:
        print('Coś poszło źle:', str(err))
    return 'Błąd'
```

Zaostrz ołówek
Rozwiązanie

1 Miałeś poprawić moduł DBcm w taki sposób, aby zgłaszał niestandardowy wyjątek ConnectionError zawsze wtedy, gdy połączenie z bazą danych działającą po stronie backendu się nie powiedzie. W kodzie modułu DBcm.py powinieneś był dodać niezbędne do tego instrukcje.

Zdefiniuj własny wyjątek jako pustą klasę, która dziedziczy po klasie Exception.

```
import mysql.connector

class ConnectionError(Exception):
    pass

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        try:
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Nowa konstrukcja try-except zabezpiecza kod odpowiedzialny za połączenie z bazą danych.

Wewnątrz pliku DBcm.py odwołuj się do wyjątków specyficznych dla bazy danych za pomocą ich pełnych nazw.

Zgłoś własny wyjątek.

2 Zmieniwszy kod modułu DBcm, miałeś użyć ołówka, aby opisać wszystkie modyfikacje, które wprowadziłbyś w przedstawionym poniżej w skrócie kodzie programu *vsearch4web.py*, aby móc skorzystać z nowo zdefiniowanego wyjątku ConnectionError.

Nie musisz tu już importować modułu mysql.connector (ponieważ moduł DBcm robi to za Ciebie).

```
from DBcm import UseDatabase, ConnectionError
~~import mysql.connector~~
...
                    the_row_titles=titles,
                    the_data=contents,)
except ~~mysql.connector.errors.InterfaceError~~ ConnectionError as err:
    print('Is your database switched on? Error:', str(err))
except Exception as err:
    print('Something went wrong:', str(err))
return 'Error'
```

Pamiętaj, aby zaimportować klasę ConnectionError z modułu DBcm.

Zmień pierwszą klauzulę except, aby przechwytywała wyjątek ConnectionError zamiast błędu InterfaceError.

Jazda próbna

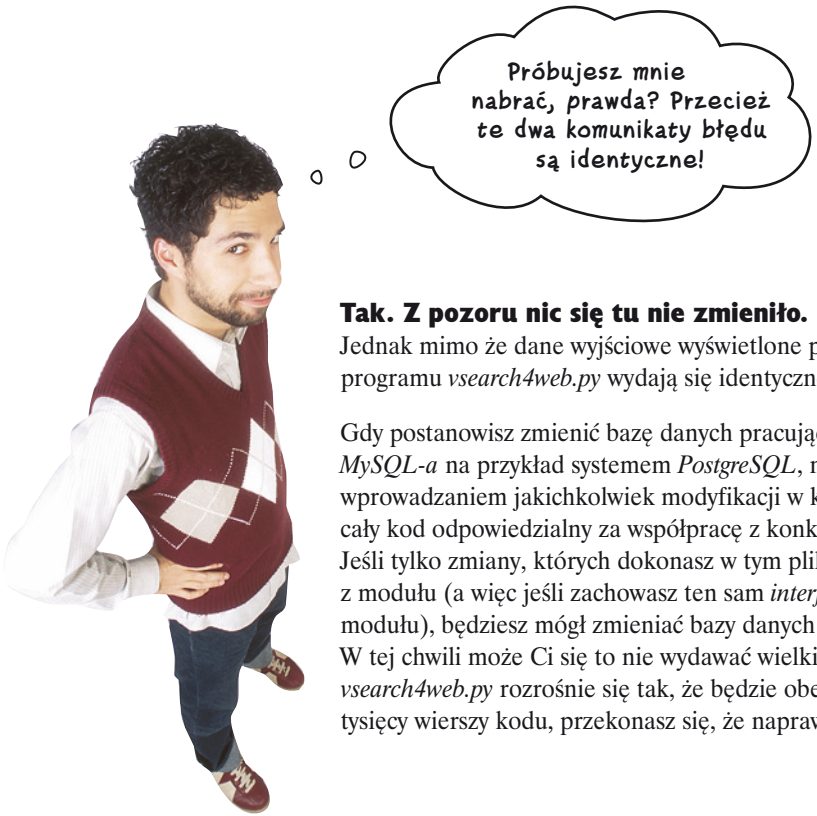
Sprawdźmy, co się zmieniło po dodaniu tego nowego kodu. Przypomnij sobie, że kod odpowiedzialny za obsługę wyjątków specyficznych dla bazy danych MySQL przeniosłeś z pliku *vsearch4web.py* do pliku *DBcm.py* (oraz zastąpiłeś go kodem przechwytującym niestandardowy wyjątek `ConnectionError`). Czy zrobiło nam to jakąś różnicę?

Poniżej znajdziesz komunikaty generowane przez poprzednią wersję programu *vsearch4web.py*, gdy bazy danych pracującej po stronie backendu nie dało się odnaleźć.

```
...
Czy Twoja baza danych jest włączona? Błąd: 2003: Can't connect to MySQL server on '127.0.0.1:3306'
(10061 Nie można nawiązać połączenia, ponieważ komputer docelowy aktywnie go odmawia)
127.0.0.1 - - [02/Jul/2017 11:05:22] "GET /viewlog HTTP/1.1" 200 -
```

Poniżej znajdziesz z kolei komunikaty generowane przez najnowszą wersję programu *vsearch4web.py*, gdy bazy danych pracującej po stronie backendu nie da się odnaleźć.

```
...
Czy Twoja baza danych jest włączona? Błąd: 2003: Can't connect to MySQL server on '127.0.0.1:3306'
(10061 Nie można nawiązać połączenia, ponieważ komputer docelowy aktywnie go odmawia)
127.0.0.1 - - [02/Jul/2017 11:13:47] "GET /viewlog HTTP/1.1" 200 -
```



Tak. Z pozoru nic się tu nie zmieniło.
Jednak mimo że dane wyjściowe wyświetlone przez bieżącą i poprzednią wersję programu *vsearch4web.py* wydają się identyczne, to, co się dzieje w tle, jest *zupełnie różne*.

Gdy postanowisz zmienić bazę danych pracującą po stronie backendu, zastępując *MySQL-a* na przykład systemem *PostgreSQL*, nie będziesz już musiał się martwić wprowadzaniem jakichkolwiek modyfikacji w kodzie programu *vsearch4web.py*, ponieważ cały kod odpowiedzialny za współpracę z konkretną bazą znajduje się w pliku *DBcm.py*. Jeśli tylko zmiany, których dokonasz w tym pliku, nie wpłyną na sposób korzystania z modułu (a więc jeśli zachowasz ten sam *interfejs*, który miały poprzednie wersje modułu), będziesz mógł zmieniać bazy danych SQL tak często, jak tylko sobie zażyczysz. W tej chwili może Ci się to nie wydawać wielkim problemem, lecz gdy kod programu *vsearch4web.py* rozrośnie się tak, że będzie obejmował setki, tysiące lub nawet dziesiątki tysięcy wierszy kodu, przekonasz się, że naprawdę stanie się to problemem.

Co jeszcze może pójść źle z modułem DBcm?

Nawet jeśli Twoja baza danych została uruchomiona i działa, nadal coś może pójść nie tak.

Na przykład dane logowania używane w celu uzyskania dostępu do bazy danych mogą się okazać nieprawidłowe. W takim przypadku metoda `__enter__` ponownie zawiedzie, zgłaszając tym razem wyjątek `connector.errors.ProgrammingError`.

Może być też tak, że zestaw kodu związany z menedżerem kontekstu `UseDatabase` zgłosi wyjątek, ponieważ nigdy nie możemy mieć pewności, że wykona się on prawidłowo. Jeśli zapytanie bazodanowe (kod SQL, który wykonujesz) zawiera błąd, zgłaszany jest *również* wyjątek `mysql.connector.errors.ProgrammingError`.

Komunikat błędu związany z błędem zapytania SQL jest inny niż ten, który dotyczy błędu w danych logowania, ale w obydwu sytuacjach zgłaszany jest ten sam wyjątek: `mysql.connector.errors.ProgrammingError`. W przeciwieństwie do błędów związanych z niewłaściwymi danymi logowania błędy występujące w zapytaniach SQL powodują wyjątek, który jest zgłaszany w czasie wykonywania instrukcji `with`. Oznacza to, że musisz rozważyć zabezpieczenie kodu przed tym wyjątkiem w więcej niż tylko jednym miejscu. Pytanie brzmi: gdzie?

Aby na nie odpowiedzieć, jeszcze raz przyjrzymy się kodowi modułu `DBcm`, który został pokazany poniżej.

```
import mysql.connector

class ConnectionError(Exception):
   pass

class UseDatabase:

   def __init__(self, config: dict) -> None:
      self.configuration = config

   def __enter__(self) -> 'cursor':
      try:
         self.conn = mysql.connector.connect(**self.configuration)
         self.cursor = self.conn.cursor()
         return self.cursor
      except mysql.connector.errors.InterfaceError as err:
         raise ConnectionError(err)

   def __exit__(self, exc_type, exc_value, exc_trace) -> None:
      self.conn.commit()
      self.cursor.close()
      self.conn.close()
```

Ten kod może spowodować zgłoszenie wyjątku ProgrammingError.

Co jednak z wyjątkami, które pojawiają się wewnątrz zestawu instrukcji with? Występują one PO zakończeniu metody __enter__, ale PRZED rozpoczęciem działania metody __exit__.

Mógłbyś się pokusić o stwierdzenie, że wyjątki zgłoszone wewnątrz zestawu instrukcji `with` powinny być obsługiwane za pomocą instrukcji `try-except` znajdującej się *w obrębie* instrukcji `with`, ale strategia ta szybko doprowadziłaby do ponownego powstania problemu ściśle sprzężonego kodu. Pomyśl jednak o tym, że gdy wyjątek jest zgłaszany wewnątrz zestawu instrukcji `with` i *nie* zostaje przechwycony, instrukcja ta przekazuje szczegóły dotyczące nieprzechwyconego wyjątku do należącej do menedżera kontekstu metody `__exit__`, w której masz możliwość zajęcia się tym problemem.

Tworzenie kolejnych własnych wyjątków

Rozbudujmy kod modułu `DBcm.py` w taki sposób, aby zgłaszał jeszcze dwa dodatkowe, niestandardowe wyjątki.

Pierwszy z nich nosi nazwę `CredentialsError` i będzie zgłaszany, gdy w metodzie `__enter__` wystąpi błąd `ProgrammingError`. Drugi nosi nazwę `SQLError` i będzie zgłaszany, gdy ten sam błąd będzie zgłaszany w metodzie `__exit__`.

Definiowanie tych dwóch nowych wyjątków jest łatwe. Aby to zrobić, po prostu dodaj dwie nowe, puste klasy wyjątków na początku pliku *DBcm.py*, tak jak zostało to przedstawione poniżej.

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLError(Exception):
    pass

class UseDatabase:
    def __init__(self, config: dict) -> None:
        self.configuration = config
            ...
```

Za pomocą dwóch dodatkowych klas definiujesz dwa nowe wyjątki.

Wyjątek `CredentialsError` może się pojawić w czasie wykonania metody `__enter__`, dlatego zmieńmy jej kod w taki sposób, aby to odzwierciedlić. Przypomnij sobie, że podanie nieprawidłowej nazwy użytkownika lub hasła dostępu do bazy MySQL może skutkować zgłoszeniem wyjątku `ProgrammingError`.

```
        ...
    try:
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor
    except mysql.connector.errors.InterfaceError as err:
        raise ConnectionError(err)
    except mysql.connector.errors.ProgrammingError as err:
        raise CredentialsError(err)

def __exit__(self, exc_type, exc_value, exc_trace) -> None:
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

Dodaj ten kod do metody `__enter__`, aby obsłużyć wszelkie problemy związane z logowaniem użytkownika bazy danych.

Wprowadzone tu zmiany w kodzie modułu `DBcm.py` powodują, że zgłasza on wyjątek `CredentialsError`, gdy z poziomu kodu podasz niewłaściwą nazwę użytkownika lub hasło dostępu do Twojej bazy danych (MySQL). Odpowiednie dostosowanie kodu programu *vsearch4web.py* będzie Twoim kolejnym zadaniem.

Czy Twoje dane logowania do bazy danych są prawidłowe?

Wprowadziwszy ostatnie zmiany w pliku *DBcm.py*, poprawmy kod programu *vsearch4web.py*, zwracając szczególną uwagę na funkcję `view_the_log`. Zanim jednak zrobimy cokolwiek w tej kwestii, dodaj klasę `CredentialsError` do listy elementów importowanych z modułu `DBcm`, która znajduje się na samym początku pliku *vsearch4web.py*, jak zostało to pokazane poniżej.

Pamiętaj, aby zaimportować swój nowy wyjątek.

```
...
from DBcm import UseDatabase, ConnectionError, CredentialsError
...
```

Po dodaniu elementu do wiersza instrukcji `import` powinieneś dodać nowy zestaw instrukcji `except` do funkcji `view_the_log`. Podobnie jak miało to miejsce w przypadku kodu obsługi wyjątku `ConnectionError`, jest to bardzo proste zadanie, co widać poniżej.

```
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
   try:
      with UseDatabase(app.config['dbconfig']) as cursor:
         _SQL = """select phrase, letters, ip, browser_string, results
                   from log"""
         cursor.execute(_SQL)
         contents = cursor.fetchall()
      titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
      return render_template('viewlog.html',
                             the_title='View Log',
                             the_row_titles=titles,
                             the_data=contents,)

   except ConnectionError as err:
      print('Czy Twoja baza danych jest włączona? Błąd:', str(err))
   except CredentialsError as err:
      print('Problemy z identyfikatorem użytkownika lub hasłem dostępu. Błąd:', str(err))
   except Exception as err:
      print('Coś poszło źle:', str(err))
   return 'Błąd'
```

Dodaj ten fragment kodu do funkcji view_the_log, aby przechwycić wyjątek spowodowany podaniem przez jej użytkownika niewłaściwej nazwy użytkownika lub hasła dostępu do bazy danych MySQL.

W gruncie rzeczy nie ma tu nic nowego, ponieważ wszystko, co robisz, sprowadza się do powtórzenia działań wykonywanych już przy okazji dodawania kodu obsługi błędu `ConnectionError`. Oczywiście, gdy spróbujesz teraz połączyć się ze swoją bazą danych, korzystając z nieprawidłowej nazwy użytkownika (lub hasła), Twoja aplikacja WWW wyświetli w oknie terminala odpowiedni komunikat, który powinien być podobny do przedstawionego poniżej.

```
...
Problemy z identyfikatorem użytkownika lub hasłem dostępu. Błąd: 1045 (28000): Access denied for user
'vsearcherror'@'localhost' (using password: YES)
127.0.0.1 - - [02/Jul/2017 20:41:18] "GET /viewlog HTTP/1.1" 200 -
```

Teraz, gdy Twój kod „wie już wszystko" o wyjątku CredentialsError, generowany jest komunikat specyficzny dla błędu.

Obsługa wyjątku SQLError odbywa się w inny sposób

Zarówno wyjątek `ConnectionError`, jak i `CredentialsError` są zgłaszane w wyniku wystąpienia problemów z wykonaniem kodu metody `__enter__`. Gdy którykolwiek z tych wyjątków zostaje zgłoszony, zestaw związany z odpowiednią instrukcją `with` **nie** jest wykonywany.

Jeśli wszystko jest w porządku, zestaw instrukcji `with` zostaje wykonany w normalny sposób.

Przypomnij sobie znajdującą się w funkcji `log_request` instrukcję `with`, w której używany jest menedżer kontekstu `UseDatabase` (zapewniany przez moduł `DBcm`) do wstawiania danych do bazy działającej po stronie backendu, tak jak pokazano to poniżej.

Powinniśmy się przejmować tym, co się stanie, gdy coś pójdzie nie tak z tym fragmentem kodu (tj. z kodem wewnątrz zestawu instrukcji with).

```
with UseDatabase(app.config['dbconfig']) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Jeśli (z jakiegoś powodu) Twój kod SQL zawiera błąd, moduł *MySQL Connector* generuje wyjątek `ProgrammingError`, podobny do tego, który jest zgłaszany w czasie wykonywania metody `__enter__` należącej do menedżera kontekstu. Ponieważ jednak wyjątek ten zdarza się *wewnątrz* Twojego menedżera kontekstu (tj. w obrębie instrukcji `with`) i *nie* jest tam przechwytywany, zostaje przekazany do metody `__exit__` pod postacią trzech argumentów: *typu* wyjątku, jego *wartości* oraz *tracebacku* związanego z wyjątkiem.

Te trzy argumenty wyjątku są gotowe do użycia.

Gdy spojrzysz na istniejący kod metody `__exit__` znajdującej się w module `DBcm`, przekonasz się, że te trzy argumenty tylko czekają na to, aby z nich skorzystać.

```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

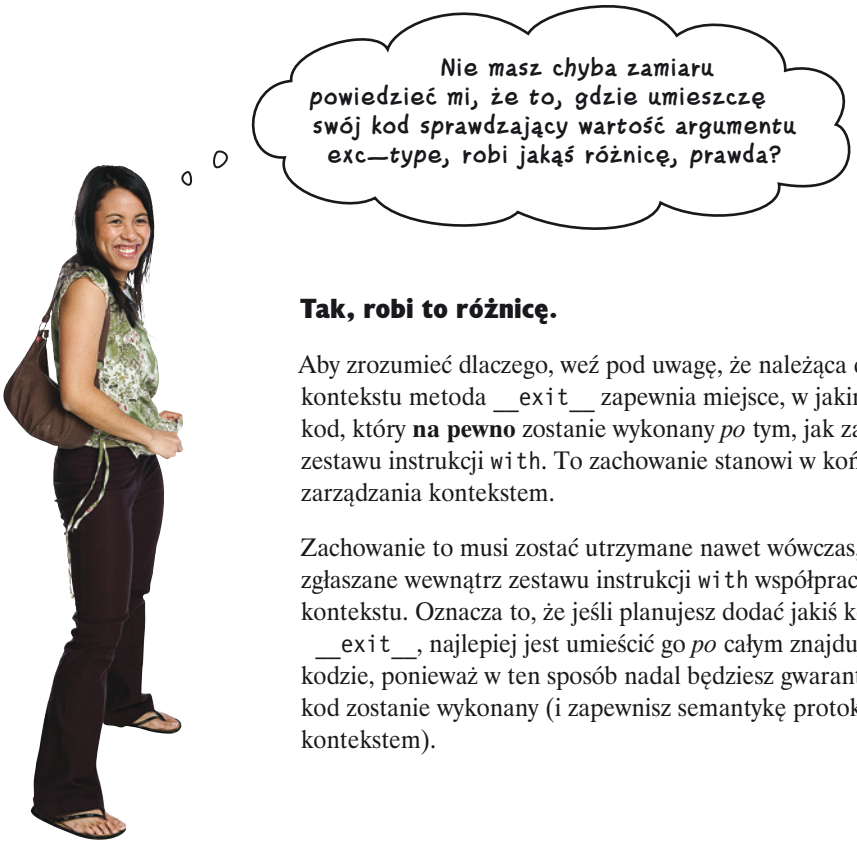
Gdy wyjątek jest zgłaszany w obrębie zestawu instrukcji `with` i nie zostaje przechwycony, menedżer kontekstu przerywa wykonywanie kodu należącego do zestawu tej instrukcji i przeskakuje do metody `__exit__`, którą następnie wykonuje. Wiedząc to, możesz napisać kod sprawdzający wyjątki interesujące z punktu widzenia Twojej aplikacji. Jeśli jednak nie zostaje zgłoszony żaden wyjątek, wszystkie nasze trzy argumenty (`exc_type`, `exc_value` oraz `exc_traceback`) mają wartość `None`. W przeciwnym razie zawierają szczegóły dotyczące zgłoszonego wyjątku.

None to wartość null w języku Python.

Skorzystajmy z tego zachowania, aby zgłosić wyjątek `SQLError`, gdy coś pójdzie źle wewnątrz instrukcji `with` kontrolowanej przez menedżer kontekstu `UseDatabase`.

Zachowaj ostrożność, wybierając miejsce dodania kodu

Aby się przekonać, czy wewnątrz instrukcji `with` pojawił się jakiś nieprzechwycony wyjątek, wewnątrz zestawu metody `__exit__` sprawdź wartość argumentu `exc_type`, zachowując przy tym dużą ostrożność w kwestii tego, gdzie umieścisz swój nowy kod.



Tak, robi to różnicę.

Aby zrozumieć dlaczego, weź pod uwagę, że należąca do menedżera kontekstu metoda `__exit__` zapewnia miejsce, w jakim możesz dodać kod, który **na pewno** zostanie wykonany *po* tym, jak zakończy się działanie zestawu instrukcji `with`. To zachowanie stanowi w końcu część protokołu zarządzania kontekstem.

Zachowanie to musi zostać utrzymane nawet wówczas, gdy wyjątki są zgłaszane wewnątrz zestawu instrukcji `with` współpracującej z menedżerem kontekstu. Oznacza to, że jeśli planujesz dodać jakiś kod do metody `__exit__`, najlepiej jest umieścić go *po* całym znajdującym się już tam kodzie, ponieważ w ten sposób nadal będziesz gwarantował, że ten istniejący kod zostanie wykonany (i zapewnisz semantykę protokołu zarządzania kontekstem).

Spójrzmy raz jeszcze na zamieszczony poniżej dotychczasowy kod metody `__exit__` w kontekście powyższych informacji. Weź pod uwagę, że kod, który dodamy, powinien zgłaszać wyjątek `SQLError`, jeśli wystąpił błąd `ProgrammingError`.

```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

Gdy dodasz kod w tym miejscu, a kod ten zgłosi wyjątek, trzy istniejące już wiersze kodu nie zostaną wykonane.

Dodanie kodu PO tych trzech istniejących wierszach kodu gwarantuje, że metoda __exit__ wykona swoje zadanie PRZED obsłużeniem jakiegokolwiek przekazanego jej wyjątku.

Zgłaszanie wyjątku SQLError

Na tym etapie masz już dodaną klasę `SQLError` w pliku *DBcm.py*, co pokazano poniżej.

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLError(Exception):
    pass

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config
        ...
```

To jest miejsce, w którym dodałeś wyjątek SQLError.

Mając zdefiniowaną klasę `SQLError`, musisz już tylko dodać trochę kodu do metody `__exit__`, za pomocą którego sprawdzisz, czy wartość argumentu `exc_type` wskazuje interesujący Cię wyjątek, a jeśli tak właśnie będzie, zgłosisz wyjątek `SQLError`. Zadanie to jest tak proste, że oparliśmy się występującemu zwykle w serii *Rusz głową!* przymusowi przekształcenia procesu tworzenia wymaganego kodu w ćwiczenie, ponieważ nikt nie chce obrażać niczyjej inteligencji na tym etapie lektury książki. Poniżej znajdziesz zatem kod, który powinieneś dołączyć do metody `__exit__`.

```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
    if exc_type is mysql.connector.errors.ProgrammingError:
        raise SQLError(exc_value)
```

Jeśli wystąpi błąd ProgrammingError, zgłoś wyjątek SQLError.

Jeśli chcesz się **dodatkowo zabezpieczyć** i zrobić coś sensownego z każdym innym, nieoczekiwanym wyjątkiem przesłanym do metody `__exit__`, na końcu możesz dodać zestaw instrukcji `elif`, który zgłosi niespodziewany wyjątek do wywołującego kodu, tak jak zostało to przedstawione poniżej.

```
    ...
    self.conn.close()
    if exc_type is mysql.connector.errors.ProgrammingError:
        raise SQLError(exc_value)
    elif exc_type:
        raise exc_type(exc_value)
```

Ta instrukcja elif zgłasza każdy inny wyjątek, który mógłby wystąpić.

Jazda próbna

Dodawszy obsługę wyjątku `SQLError` do pliku *DBcm.py*, dołóż kolejny zestaw klauzuli `except` do funkcji `view_the_log`, aby przechwytywać wszystkie błędy typu `SQLError`, jakie się pojawiają.

Dodaj ten kod do funkcji view_the_log znajdującej się w aplikacji vsearch4web.py.

```
except ConnectionError as err:
   print('Czy Twoja baza danych jest włączona? Błąd:', str(err))
except CredentialsError as err:
   print('Problemy z identyfikatorem użytkownika lub hasłem dostępu. Błąd:', str(err))
except SQLError as err:
   print('Czy Twoje zapytanie jest poprawne? Błąd:', str(err))
except Exception as err:
   print('Coś poszło źle:', str(err))
return 'Błąd'
```

Do listy elementów importowanych z modułu `DBcm` nie zapomnij też dodać swojej nowej klasy wyjątku.

```
from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLError
```

Gdy zapiszesz już swój plik *vsearch4web.py*, Twoja aplikacja WWW powinna się przeładować i być gotowa do testów. Jeśli spróbujesz wykonać zapytanie SQL zawierające błędy, odpowiedni wyjątek zostanie obsłużony przez powyższy kod, tak jak zostało to pokazane poniżej.

```
...
Czy Twoje zapytanie jest poprawne? Błąd: 1146 (42S02): Table 'vsearchlogdb.logerror' doesn't exist
127.0.0.1 - - [03/Jul/2017 10:02:58] "GET /viewlog HTTP/1.1" 200 -
```

Nie występują tu już ogólne wyjątki ProgrammingError pochodzące z modułu MySQL Connector, ponieważ Twój niestandardowy kod obsługi wyjątków przechwytuje te błędy.

Podobnie, gdy stanie się coś niespodziewanego, kod przechwytujący wszystkie wyjątki, który dodaliśmy do Twojej aplikacji, wkracza do akcji, wyświetlając odpowiedni komunikat.

```
...
Coś poszło źle: Jakiś nieznany wyjątek.
127.0.0.1 - - [03/Jul/2017 10:09:27] "GET /viewlog HTTP/1.1" 200 -
```

Gdy dzieje się coś nieoczekiwanego, Twój kod obsługuje również ten problem.

Gdy dodasz już do swojej aplikacji WWW kod obsługi wyjątków, będzie ona kontynuowała swoje działanie bez wyświetlania użytkownikom przerażającej lub dezinformującej strony błędu, niezależnie od tego, jaki błąd czasu wykonania wystąpi.

A naprawdę fajną rzeczą jest tutaj to, że kod ten przekształca ogólny wyjątek ProgrammingError zapewniany przez moduł MySQL Connector w dwa niestandardowe wyjątki, które mają określone znaczenie dla naszej aplikacji WWW.

Tak, to prawda. I daje nam to spore możliwości.

Szybkie przypomnienie: poprawianie odporności

Poświęćmy chwilę na przypomnienie sobie, co mieliśmy do zrobienia w tym rozdziale. Podejmując próbę uczynienia naszej aplikacji WWW bardziej odporną na błędy, musieliśmy odpowiedzieć na cztery pytania dotyczące czterech zidentyfikowanych przez nas problemów. Przejrzyjmy teraz te pytania i opiszmy, co udało się nam zrobić.

1. **Co się stanie, gdy połączenie z bazą danych zawiedzie?**
   Utworzyłeś nowy wyjątek o nazwie `ConnectionError`, który jest zgłaszany, gdy bazy danych działającej po stronie backendu nie da się znaleźć. Następnie użyłeś konstrukcji `try-except`, aby obsłużyć ewentualne wystąpienie tego wyjątku.

2. **Czy Twoja aplikacja jest zabezpieczona przed atakami sieciowymi?**
   Dzięki „szczęśliwemu przypadkowi" polegającemu na wybraniu frameworka *Flask* oraz silnika szablonów *Jinja2*, a także specyfikacji DB-API zapewnianego przez język Python nasza aplikacja WWW jest zabezpieczona przed najczęściej występującymi atakami sieciowymi. A zatem, tak, Twoja aplikacja jest chroniona przed *niektórymi* atakami (jednak nie przed wszystkimi).

3. **Co się stanie, gdy coś będzie długo trwało?**
   Nadal nie odpowiedzieliśmy na to pytanie, a jedynie pokazaliśmy, co się dzieje, gdy Twoja aplikacja potrzebuje 15 sekund, aby odpowiedzieć na żądanie użytkownika. W takiej sytuacji musi on czekać (lub, co bardziej prawdopodobne, ma dość oczekiwania i opuszcza stronę).

4. **Co się stanie, gdy wywołanie funkcji się nie uda?**
   Do zabezpieczenia wywołania funkcji zastosowałeś konstrukcję `try-except`, która zapewniła Ci kontrolę nad tym, co użytkownik Twojej aplikacji WWW widzi, gdy coś idzie nie tak.

Co się dzieje, gdy coś trwa długo?

Pytanie to pojawiło się, gdy podczas przeprowadzania naszego pierwszego ćwiczenia na początku tego rozdziału sprawdzaliśmy wywołania funkcji `cursor.execute` znajdujące się w funkcjach `log_request` oraz `view_the_log`. Mimo że pracowałeś już nad obydwoma tymi funkcjami, starając się odpowiedzieć na przedstawione powyżej pytania 1. i 4., w gruncie rzeczy nadal masz tu jeszcze coś do zrobienia.

Obydwie funkcje, `log_request` i `view_the_log`, wykorzystują menedżer kontekstu `UseDatabase` do wykonania zapytania SQL. Funkcja `log_request` **zapisuje** w bazie danych szczegóły związane z przeprowadzanym wyszukiwaniem, a funkcja `view_the_log` **odczytuje** dane z tej bazy.

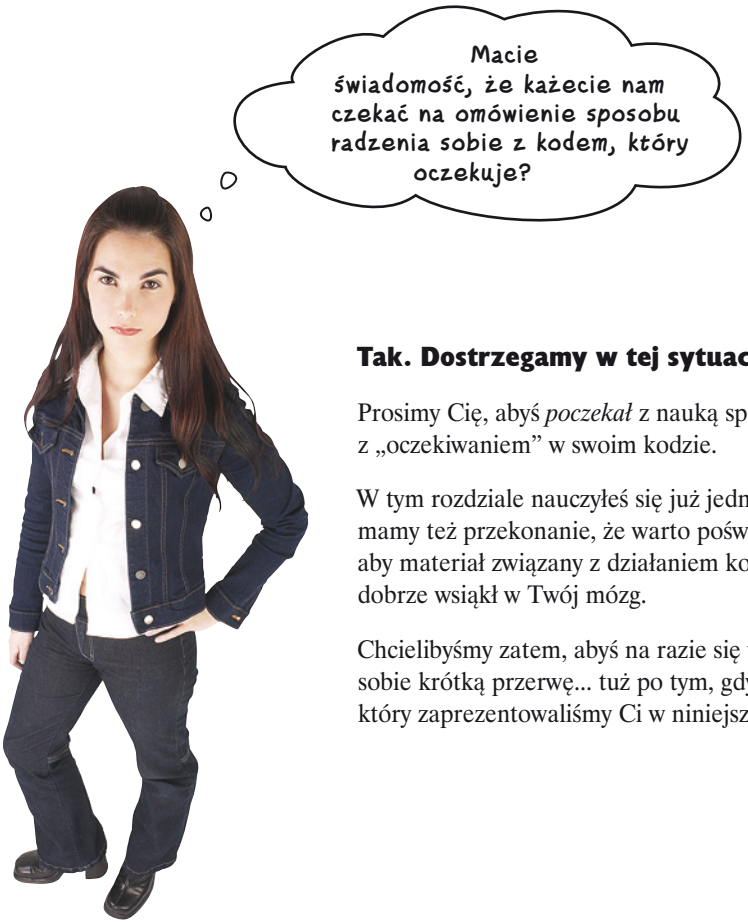
Powstaje w związku z tym pytanie: *co powinieneś zrobić, gdy ten zapis lub odczyt wymaga dużo czasu?*

Cóż, podobnie jak w przypadku wielu kwestii w świecie programowania komputerów, odpowiedź brzmi: to zależy.

Jak radzić sobie z czekaniem? To zależy...

Sposób, w jaki postanowisz radzić sobie z kodem, który zmusza użytkowników Twojej aplikacji do oczekiwania — czy to na operacje odczytu, czy też zapisu — może okazać się dość skomplikowany. Z tego powodu zamierzamy wstrzymać się z omawianiem tej kwestii i odłożymy prezentację rozwiązania do czasu rozpoczęcia kolejnego, krótkiego rozdziału.

W gruncie rzeczy następny rozdział jest tak krótki, że nie zasługuje na osobny numer (jak się już wkrótce przekonasz), jednak zaprezentowany w nim materiał jest wystarczająco skomplikowany, aby usprawiedliwić wydzielenie go z bieżącego rozdziału, którego tematem przewodnim jest zapewniany przez Pythona mechanizm `try-except`. Zatem wstrzymajmy się na jakiś czas, zanim powrócimy do pytania dotyczącego problemu nr 3: *co się stanie, gdy coś zabiera dużo czasu?*



Tak. Dostrzegamy w tej sytuacji pewną ironię.

Prosimy Cię, abyś *poczekał* z nauką sposobu radzenia sobie z „oczekiwaniem” w swoim kodzie.

W tym rozdziale nauczyłeś się już jednak bardzo dużo, mamy też przekonanie, że warto poświęcić trochę czasu, aby materiał związany z działaniem konstrukcji `try-except` dobrze wsiąkł w Twój mózg.

Chcielibyśmy zatem, abyś na razie się wstrzymał i zrobił sobie krótką przerwę... tuż po tym, gdy przejrzysz kod, który zaprezentowaliśmy Ci w niniejszym rozdziale.

Kod przykładów z rozdziału 11. (1. z 3)

To jest zawartość pliku try_example.py.

```
try:
   with open('myfile.txt') as fh:
      file_data = fh.read()
   print(file_data)
except FileNotFoundError:
   print('Brak pliku z danymi.')
except PermissionError:
   print('Brak wymaganych uprawnień.')
except Exception as err:
   print('Wystąpił jakiś inny błąd:', str(err))
```

```
import mysql.connector

class ConnectionError(Exception):
   pass

class CredentialsError(Exception):
   pass

class SQLError(Exception):
   pass

class UseDatabase:
   def __init__(self, config: dict) -> None:
      self.configuration = config

   def __enter__(self) -> 'cursor':
      try:
         self.conn = mysql.connector.connect(**self.configuration)
         self.cursor = self.conn.cursor()
         return self.cursor
      except mysql.connector.errors.InterfaceError as err:
         raise ConnectionError(err)
      except mysql.connector.errors.ProgrammingError as err:
         raise CredentialsError(err)

   def __exit__(self, exc_type, exc_value, exc_trace) -> None:
      self.conn.commit()
      self.cursor.close()
      self.conn.close()
      if exc_type is mysql.connector.errors.ProgrammingError:
         raise SQLError(exc_value)
      elif exc_type:
         raise exc_type(exc_value)
```

To jest wersja pliku DBcm.py odporna na wyjątki.

Kod przykładów z rozdziału 11. (2. z 3)

To jest wersja programu vsearch4web.py, która każe nam czekać...

```
from flask import Flask, render_template, request, escape, session

from vsearch import search4letters

from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLError
from checker import check_logged_in

from time import sleep

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
   session['logged_in'] = True
   return 'Teraz jesteś zalogowany.'

@app.route('/logout')
def do_logout() -> str:
   session.pop('logged_in')
   return 'Teraz jesteś wylogowany.'

def log_request(req: 'flask_request', res: str) -> None:
   sleep(15)
   with UseDatabase(app.config['dbconfig']) as cursor:
      _SQL = """insert into log
                (phrase, letters, ip, browser_string, results)
                values
                (%s, %s, %s, %s, %s)"""
      cursor.execute(_SQL, (req.form['phrase'],
                            req.form['letters'],
                            req.remote_addr,
                            req.user_agent.browser,
                            res, ))

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
```

Dobrym pomysłem będzie z pewnością zabezpieczenie tej instrukcji with w podobny sposób, w jaki chroniliśmy instrukcję with znajdującą się w funkcji view_the_log (przedstawionej na następnej stronie).

Pozostała część funkcji do_search znajduje się na początku następnej strony.

Kod przykładów z rozdziału 11. (3. z 3)

```
    results = str(search4letters(phrase, letters))
    try:
       log_request(request, results)
    except Exception as err:
       print('***** Logowanie się nie powiodło; wystąpił błąd:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Witamy na stronie internetowej search4letters!')

@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
       with UseDatabase(app.config['dbconfig']) as cursor:
          _SQL = """select phrase, letters, ip, browser_string, results
                    from logerror"""
          cursor.execute(_SQL)
          contents = cursor.fetchall()
       # raise Exception("Jakiś nieznany wyjątek.")
       titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
       return render_template('viewlog.html',
                              the_title='View Log',
                              the_row_titles=titles,
                              the_data=contents,)

    except ConnectionError as err:
       print('Czy Twoja baza danych jest włączona? Błąd:', str(err))
    except CredentialsError as err:
       print('Problemy z identyfikatorem użytkownika lub hasłem dostępu. Błąd:', str(err))
    except SQLError as err:
       print('Czy Twoje zapytanie jest poprawne? Błąd:', str(err))
    except Exception as err:
       print('Coś poszło źle:', str(err))
    return 'Błąd'

app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
     app.run(debug=True)
```

To jest reszta kodu funkcji do_search.

Radzenie sobie z oczekiwaniem

Twój kod może nieraz potrzebować sporo czasu, aby się wykonać.

Zależnie od tego, kto dokonuje obserwacji, to może, lecz nie musi być problemem. Jeśli wykonanie jakiegoś kodu wymaga 30 sekund, lecz odbywa się „w tle", nawet tak długie oczekiwanie wcale nie musi być kłopotliwe. Jeśli jednak użytkownik aplikacji czeka na jej odpowiedź pół minuty, nie ma takiej osoby, która mogłaby tego nie zauważyć. Sposób, w jaki powinieneś naprawić tego rodzaju problem, zależy od tego, co próbujesz robić (i kto musi czekać). W tym krótkim rozdziale pokrótce opiszemy niektóre możliwości, a następnie przyjrzymy się jednemu rozwiązaniu problemu, z którym mamy do czynienia, czyli odpowiemy na pytanie: *co się stanie, gdy coś zabiera dużo czasu?*

Oczekiwanie: co robić?

Gdy piszesz kod, który potencjalnie może zmuszać użytkowników do czekania, powinieneś starannie przemyśleć, co tak naprawdę chcesz zrobić. Rozważmy kilka punktów widzenia.

Być może mamy tu do czynienia z przypadkiem, w którym oczekiwanie na zapis *różni się* od oczekiwania na odczyt, zwłaszcza że ma to związek z tym, jak działa Twoja aplikacja WWW?

Przyjrzyjmy się raz jeszcze zapytaniom SQL znajdującym się w funkcjach `log_request` oraz `view_the_log`, aby przekonać się, w jaki sposób z nich korzystasz.

Jak odpytujesz swoją bazę danych?

W funkcji `log_request` korzystamy z instrukcji `INSERT` języka SQL, aby dodać szczegóły związane z żądaniem do bazy danych działającej po stronie backendu. Gdy funkcja ta jest wywoływana, **oczekuje** na wykonanie instrukcji `INSERT` za pomocą metody `cursor.execute`.

```
def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

W tym punkcie aplikacja WWW „blokuje się", oczekując, aż baza danych zrobi, co do niej należy.

Porady maniaka

Kod oczekujący, aż skończy się jakieś zewnętrzne działanie, jest określany mianem „kodu blokującego", ponieważ wykonanie Twojego programu zostaje **zablokowane** do czasu zakończenia tego oczekiwania. Ogólnie rzecz biorąc, kod blokujący, który wymaga straty zauważalnej ilości czasu, jest zły.

Tak samo jest z funkcją `view_the_log`, która również oczekuje na wykonanie instrukcji `SELECT` języka SQL.

```
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                      from logerror"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)

    except ConnectionError as err:
        ...
```

Twoja aplikacja WWW „blokuje się" również tutaj, czekając na bazę danych.

Aby zaoszczędzić miejsce, nie pokazujemy tu całości kodu funkcji view_the_log. Kod odpowiedzialny za obsługę wyjątków w dalszym ciągu znajduje się w tym miejscu.

Obydwie funkcje blokują aplikację. Przyjrzyj się jednak bliżej temu, co się dzieje *po* wywołaniu metody `cursor.execute` w obydwu funkcjach. W funkcji `log_request` wywołanie tej metody jest ostatnią wykonywaną operacją, a w funkcji `view_the_log` wynik wywołania metody `cursor.execute` jest używany w dalszej części kodu tej funkcji.

Rozważmy implikacje tej różnicy.

Instrukcje INSERT i SELECT są różne

Jeśli czytając tytuł tego podrozdziału, pomyślałeś sobie „Oczywiście, że są!”, zapewniamy Cię (co prawda na dość zaawansowanym etapie lektury tej książki), że nie postradaliśmy zmysłów.

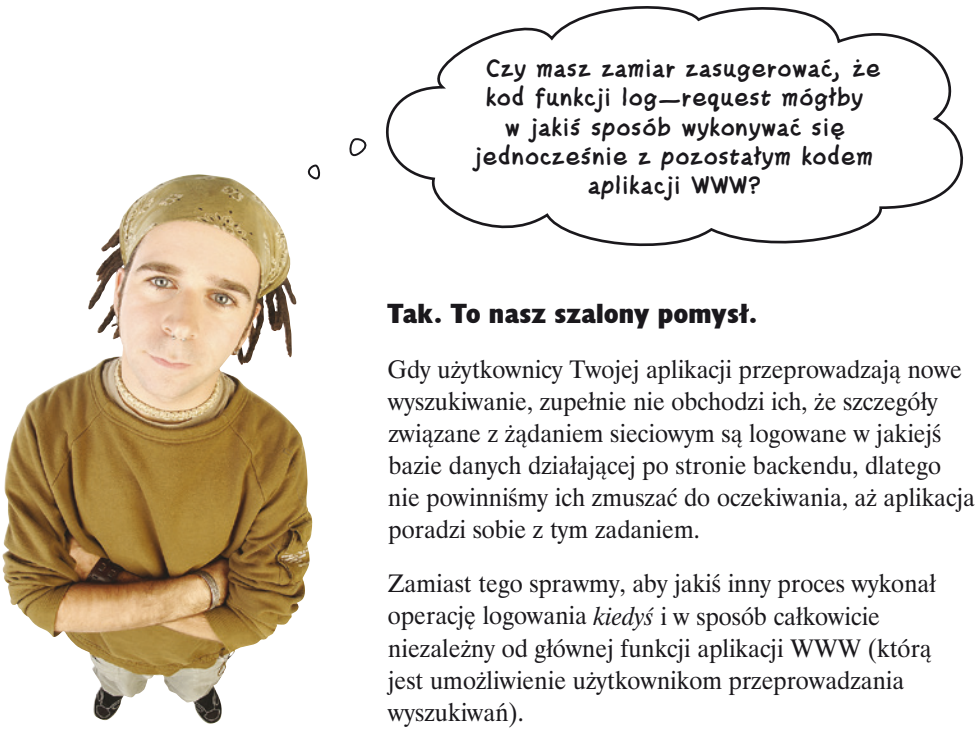
Tak, instrukcja `INSERT` języka SQL *różni się* od jego instrukcji `SELECT`, ale jeśli chodzi o Twoje zastosowanie obydwu tych instrukcji w aplikacji WWW, okazuje się, że instrukcja `INSERT` występująca w funkcji `log_request` wcale nie musi jej blokować, podczas gdy instrukcja `SELECT` należąca do funkcji `view_the_log` musi to robić, co sprawia, że zapytania zawierające te instrukcje są *zupełnie* inne.

Spostrzeżenie to ma kluczowe znaczenie.

Gdyby znajdująca się w funkcji `view_the_log` instrukcja `SELECT` nie oczekiwała na dane zwrócone z bazy danych działającej po stronie backendu, kod występujący po wywołaniu metody `cursor.execute` prawdopodobnie by zawiódł (ponieważ nie miałby danych, na których powinien operować). Funkcja `view_the_log` **musi** blokować aplikację, ponieważ musi czekać na dane, zanim będzie mogła kontynuować swoje działanie.

Gdy w swojej aplikacji WWW wywołujesz funkcję `log_request`, chcesz zalogować w bazie danych szczegóły związane z bieżącym żądaniem sieciowym. Tak naprawdę nie obchodzi Cię, *kiedy* do tego dojdzie, chodzi Ci tylko o to, aby się to stało. Funkcja `log_request` nie zwraca wartości ani danych; kod wywołujący nie musi czekać na odpowiedź. Wszystko, o co tu dbamy, sprowadza się do tego, aby żądanie sieciowe zostało *ostatecznie* zalogowane.

To z kolei prowadzi do pytania: dlaczego funkcja `log_request` zmusza kod wywołujący do oczekiwania?



Tak. To nasz szalony pomysł.

Gdy użytkownicy Twojej aplikacji przeprowadzają nowe wyszukiwanie, zupełnie nie obchodzi ich, że szczegóły związane z żądaniem sieciowym są logowane w jakiejś bazie danych działającej po stronie backendu, dlatego nie powinniśmy ich zmuszać do oczekiwania, aż aplikacja poradzi sobie z tym zadaniem.

Zamiast tego sprawmy, aby jakiś inny proces wykonał operację logowania *kiedyś* i w sposób całkowicie niezależny od głównej funkcji aplikacji WWW (którą jest umożliwienie użytkownikom przeprowadzania wyszukiwań).

Robienie więcej niż jednej rzeczy naraz

Oto plan: sprawisz, że funkcja `log_request` będzie się wykonywała niezależnie od głównego kodu Twojej aplikacji WWW. Aby to zrobić, będziesz musiał zmienić go w taki sposób, aby każde wywołanie funkcji `log_request` odbywało się równolegle. Będzie to oznaczało, że aplikacja nie będzie już musiała oczekiwać na zakończenie działania tej funkcji przed obsłużeniem kolejnego żądania od innego użytkownika (a więc nie będziemy tu już mieli do czynienia z opóźnieniami).

Dzięki temu rozwiązaniu nie będzie nas (ani użytkowników) już obchodziło, czy wykonanie funkcji `log_request` odbywa się natychmiast, czy też zajmuje kilka sekund, minutę bądź nawet godziny. Będzie Cię interesować jedynie to, że kod ten w końcu zostanie wykonany.

Równoległy kod: masz wybór

Jeśli chodzi o równoległe wykonywanie kodu Twojej aplikacji, język Python oferuje Ci kilka możliwości. Oprócz wielu rozwiązań zapewnianych przez moduły niezależne pomoc w postaci wbudowanych funkcji oferuje Ci tu biblioteka standardowa.

Pełną listę (oraz szczegółowy opis) możliwości oferowanych przez bibliotekę standardową Pythona w zakresie programowania równoległego znajdziesz pod adresem *https://docs.python.org/3/library/concurrency.html.*

Jedną z najbardziej znanych jest biblioteka `threading` zapewniająca wysokopoziomowy interfejs dla implementacji mechanizmu wątków dostarczanego przez system operacyjny, pod którego kontrolą działa Twoja aplikacja WWW. Aby móc korzystać z tej biblioteki, na początku kodu swojego programu musisz jedynie zaimportować klasę `Thread` z modułu `threading`, tak jak pokazano to poniżej.

```
from threading import Thread
```

Śmiało, przejdź teraz do kodu programu *vsearch4web.py* i dodaj w nim ten wiersz.

Teraz zaczyna się prawdziwa zabawa.

Aby utworzyć nowy wątek, powinieneś utworzyć obiekt klasy `Thread`, przypisując nazwę funkcji, która ma być przez niego wykonywana, do argumentu o nazwie `target` oraz przekazując wszystkie jej argumenty w postaci krotki za pomocą argumentu o nazwie `args`. Utworzony w ten sposób obiekt powinien następnie zostać przypisany do dowolnie wybranej przez Ciebie zmiennej.

Załóżmy na przykład, że masz funkcję o nazwie `execute_slowly`, która przyjmuje trzy argumenty będące liczbami. W kodzie wywołującym tę funkcję te trzy wartości zostały przypisane do zmiennych o nazwach `glacial`, `plodding` oraz `leaden`. Poniżej możesz zobaczyć, w jaki sposób funkcja `execute_slowly` jest wywoływana normalnie (tj. wtedy, gdy nie interesuje nas jej równoległe wykonywanie).

```
execute_slowly(glacial, plodding, leaden)
```

Jeśli wykonanie funkcji `execute_slowly` zajmuje 30 sekund, wywołujący ją kod blokuje działanie i oczekuje przez pół minuty, zanim będzie w stanie zrobić cokolwiek innego. Porażka.

Nie daj się zdołować: użyj wątków

W ogólnym rozrachunku czekanie 30 sekund na zakończenie wykonania funkcji `execute_slowly` nie wydaje się oznaczać końca świata. Jeśli jednak użytkownik Twojej aplikacji siedzi i czeka, na pewno zacznie się zastanawiać, co poszło nie tak.

Jeśli Twoja aplikacja może kontynuować działanie w czasie, gdy funkcja `execute_slowly` robi swoje, możesz utworzyć obiekt klasy `Thread`, aby wykonać tę funkcję w sposób równoległy. Poniżej raz jeszcze możesz zobaczyć zwykłe wywołanie funkcji `execute_slowly` wraz z kodem, który powoduje przekształcenie jej wywołania w żądanie wykonania w ramach wątku.

To jest oryginalne wywołanie funkcji.

```
execute_slowly(glacial, plodding, leaden)
```

Zaimportuj wymagany moduł i klasę w okolicach początku swojego kodu...

```
from threading import Thread
...
t = Thread(target=execute_slowly, args=(glacial, plodding, leaden))
```

...a następnie utwórz nowy obiekt klasy Thread, który otrzymuje nazwę funkcji do wykonania, jak również wartości wszystkich argumentów, których się ona spodziewa.

Jasne, to użycie klasy `Thread` wygląda trochę dziwnie, ale w rzeczywistości takie nie jest. Kluczem do zrozumienia tego, co się tu dzieje, jest zauważenie, że obiekt tej klasy został przypisany do zmiennej (którą w naszym przykładzie jest `t`) oraz że funkcja `execute_slowly` musi się jeszcze wykonać.

Przypisanie obiektu klasy `Thread` do zmiennej `t` *przygotowuje* tę funkcję do wykonania. Aby poprosić mechanizm wątków języka Python o wykonanie funkcji `execute_slowly`, powinieneś uruchomić wątek w taki sposób, jak zostało to pokazane poniżej.

```
t.start()
```

Gdy wywołujesz metodę start, wykonanie funkcji związanej z wątkiem przypisanym do zmiennej t zostaje zaplanowane przez moduł threading.

W tym momencie kod wywołujący metodę `t.start` kontynuuje swoje działanie. 30-sekundowe opóźnienie związane z wykonaniem funkcji `execute_slowly` nie ma żadnego wpływu na ten kod, ponieważ wykonanie to jest obsługiwane przez moduł `threading` języka Python, a nie przez Ciebie. Moduł ten „spiskuje” z interpreterem, aby *ostatecznie* wykonać funkcję `execute_slowly`.

Zaostrz ołówek

Wywołanie funkcji `log_request` w Twojej aplikacji WWW odbywa się tylko w jednym miejscu i tylko tam powinieneś go szukać; miejscem tym jest funkcja `do_search`. Przypomnij sobie, że umieściłeś już to wywołanie wewnątrz konstrukcji `try-except`, aby zabezpieczyć się przed wystąpieniem nieoczekiwanych błędów czasu wykonania.

Zwróć również uwagę, że do kodu funkcji `log_request` dodałeś 15-sekundowe opóźnienie — korzystając w tym celu z funkcji `sleep(15)` — aby spowolnić jej działanie. Poniżej został przedstawiony aktualny kod funkcji `do_search`.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logowanie się nie powiodło; wystąpił błąd:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Oto jak wywołujesz funkcję log_request w tej chwili.

Założymy, że dodałeś już wiersz `from threading import Thread` na początku kodu swojej aplikacji WWW.

Weź do ręki ołówek i w pustym miejscu pozostawionym poniżej napisz kod, który wstawiłbyś do funkcji `do_search` zamiast standardowego wywołania funkcji `log_request`.

Pamiętaj, że powinieneś użyć obiektu klasy `Thread`, aby uruchomić funkcję `log_request`, tak jak robiliśmy to z funkcją `execute_slowly` w przykładzie przedstawionym na poprzedniej stronie.

Dodaj kod związany z wątkiem, którego użyłbyś, aby ostatecznie wykonać funkcję log_request.

.....

.....

.....

Zaostrz ołówek
Rozwiązanie

Wywołanie funkcji `log_request` w Twojej aplikacji WWW odbywa się tylko w jednym miejscu i tylko tam powinieneś go szukać; miejscem tym jest funkcja `do_search`. Przypomnij sobie, że umieściłeś już to wywołanie wewnątrz konstrukcji `try-except`, aby zabezpieczyć się przed wystąpieniem nieoczekiwanych błędów czasu wykonania.

Zwróć również uwagę, że do kodu funkcji `log_request` dodałeś 15-sekundowe opóźnienie — korzystając w tym celu z funkcji `sleep(15)` — aby spowolnić jej działanie. Poniżej przedstawiony został aktualny kod funkcji `do_search`.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logowanie się nie powiodło; wystąpił błąd:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Oto jak wywołujesz funkcję log_request w tej chwili.

Założyliśmy, że dodałeś już wiersz `from threading import Thread` na początku kodu swojej aplikacji WWW.

W pustym miejscu pozostawionym poniżej miałeś napisać kod, który wstawiłbyś do funkcji `do_search` zamiast standardowego wywołania funkcji `log_request`.

Powinieneś był użyć obiektu klasy `Thread`, aby uruchomić funkcję `log_request`, tak jak robiliśmy to z funkcją `execute_slowly` w poprzednim przykładzie.

Zostawiamy instrukcję try w spokoju (na razie).

```
try:
    t = Thread(target=log_request, args=(request, results))
    t.start()
except ...
```

Dokładnie tak jak we wcześniejszym przykładzie, wskaż funkcję, którą należy uruchomić, podaj wszystkie argumenty, których potrzebuje, i nie zapomnij o zaplanowaniu uruchomienia swojego wątku.

Zestaw klauzuli except pozostaje bez zmian, dlatego nie pokazujemy go tutaj.

Jazda próbna

Wprowadziwszy te zmiany w kodzie programu *vsearch4web.py*, jesteś gotów do kolejnego testowego uruchomienia swojej aplikacji. Spodziewasz się tu zaobserwować niemal całkowity brak opóźnienia po uruchomieniu wyszukiwania na odpowiedniej stronie (jako że kod funkcji `log_request` jest teraz wykonywany równolegle przez moduł `threading`).

Śmiało, uruchom aplikację i zobacz, co się stanie.

Rzecz jasna, gdy tylko klikniesz przycisk *Zrób to!*, Twoja aplikacja WWW natychmiast zwróci wyniki wyszukiwania. Założenie jest takie, że za wykonanie funkcji `log_request` odpowiada teraz moduł `threading`, i oczekuje on tak długo, jak tylko trzeba, aby jej działanie się zakończyło (trwa to około 15 sekund, jeśli oczywiście w kodzie funkcji umieściłeś wywołanie funkcji `sleep(15)`).

Już masz zamiar poklepać się po plecach (za dobrze wykonaną robotę), gdy — zupełnie nie wiadomo dlaczego — po około 15 sekundach w oknie terminala, w którym uruchomiłeś swoją aplikację, pojawiają się jakieś dziwne komunikaty błędów podobne do przedstawionych poniżej.

Spójrz na ten komunikat.

Ostatnie żądanie się powiodło.

```
      ...
127.0.0.1 - - [04/Jul/2017 20:56:37] "POST /search4 HTTP/1.1" 200 -
Exception in thread Thread-2:
Traceback (most recent call last):
  File "vsearch4web.py", line 35, in log_request
    cursor.execute(_SQL, (req.form['phrase'],
  File "C:\Users\lsuma\AppData\Local\Programs\Python\Python36\lib\site-packages\werkzeug\local.py", line 347,
in __getattr__
      ...
    raise RuntimeError(_request_ctx_err_msg)
RuntimeError: Working outside of request context.

This typically means that you attempted to use functionality that needed
an active HTTP request.  Consult the documentation on testing for
information about how to avoid this problem.

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "C:\Users\lsuma\AppData\Local\Programs\Python\Python36\lib\threading.py", line 916, in _bootstrap_inner
    self.run()
      ...
RuntimeError: Working outside of request context.

This typically means that you attempted to use functionality that needed
an active HTTP request.  Consult the documentation on testing for
information about how to avoid this problem.
```

Ups! Nieprzechwycony wyjątek.

Tutaj jest o wiele (!!!) więcej komunikatów tracebacku.

I kolejny... Jejku!

Gdy sprawdzisz swoją bazę danych, przekonasz się, że szczegóły dotyczące Twojego żądania sieciowego **nie** zostały zalogowane. Z powyższych komunikatów wynika, że moduł `threading` ma jakiś problem z Twoim kodem. Większość komunikatów śladu należących do drugiej grupy odnosi się do modułu `threading.py`, podczas gdy te z pierwszej grupy odnoszą się do kodu należącego do folderów *werkzeug* oraz *flask*. Jasne jest, że dodanie kodu odpowiedzialnego za obsługę wątków spowodowało powstanie **ogromnego bałaganu**. Co się tu dzieje?

Najpierw najważniejsze: nie panikuj

Twoim pierwszym odruchem może być chęć wycofania dodanego kodu, aby uruchomić funkcję `log_request` w osobnym wątku (i powrócić do znanego, dobrego stanu). Nie panikuj jednak i **nie** rób tak. Zamiast tego przyjrzyjmy się przedstawionemu poniżej akapitowi z opisem, który dwukrotnie pojawił się w komunikatach tracebacku.

```
    ...
This typically means that you attempted to use functionality that needed
an active HTTP request.  Consult the documentation on testing for
information about how to avoid this problem[1].
    ...
```

Ten komunikat pochodzi z frameworka Flask, nie z modułu `threading`. Wiemy to, ponieważ moduł ten zupełnie nie dba o to, do czego go używasz, absolutnie nie interesuje go również to, co próbujesz robić z protokołem HTTP.

Przyjrzyjmy się raz jeszcze kodowi odpowiedzialnemu za zaplanowanie wykonania naszego wątku, o którym wiemy, że potrwa 15 sekund, ponieważ właśnie tyle czasu wymaga funkcja `log_request`, aby zakończyć swoje działanie. Patrząc na przedstawiony poniżej kod, zastanów się, co się dzieje przez te 15 sekund.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
   results = str(search4letters(phrase, letters))
   try:
      t = Thread(target=log_request, args=(request, results))
      t.start()
   except Exception as err:
      print('***** Logowanie się nie powiodło; wystąpił błąd:', str(err))
   return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)
```

Co się dzieje podczas 15 sekund, których wymaga wykonanie tego wątku?

Natychmiast po zaplanowaniu wykonania wątku wywołujący go kod (czyli funkcja `do_search`) kontynuuje swoje wykonanie. Odbywa się tu (w okamgnieniu) wykonanie funkcji `render_template`, po czym funkcja `do_search` *kończy* swoje działanie.

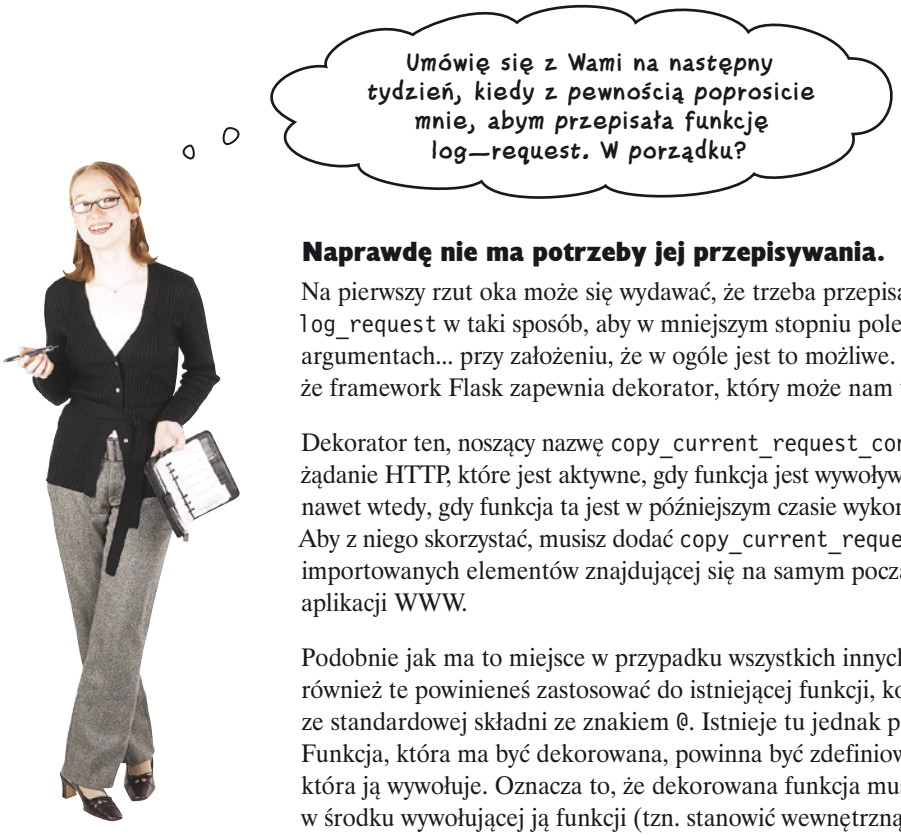
Gdy to nastąpi, wszystkie dane związane z tą funkcją (czyli jej *kontekst*) są zwalniane przez interpreter. Zmienne `request`, `phrase`, `letters`, `title` oraz `results` przestają istnieć. Jednak zmienne `request` i `results` są przekazywane jako argumenty funkcji `log_request`, która próbuje się do nich odwołać 15 sekund później. Na nieszczęście w tym momencie zmienne te już nie istnieją, ponieważ funkcja `do_search` zakończyła swoje działanie. Porażka.

[1] Oryginalny komunikat wyświetlony w języku angielskim można przetłumaczyć na polski w następujący sposób: „Zwykle oznacza to, że próbowałeś użyć funkcji, która wymagała aktywnego żądania HTTP. W dokumentacji dotyczącej testowania poszukaj informacji na temat sposobów uniknięcia tego problemu” — *przyp. tłum.*

Nie daj się zdołować: pomóc może Ci framework Flask

Na podstawie tego, czego się właśnie dowiedziałeś, wydaje się, że funkcja `log_request` (gdy jest wykonywana w ramach wątku) nie może już „widzieć” danych stanowiących jej argumenty. Wynika to z faktu, że interpreter zdążył już dawno posprzątać po sobie i zwolnił pamięć używaną przez odpowiednie zmienne (ponieważ `do_search` zakończyła swoje działanie). Konkretnie obiekt `request` nie jest już aktywny i gdy funkcja `log_request` próbuje go odszukać, nie może zostać znaleziony.

Co więc można zrobić? Nie martw się: rozwiązanie jest na wyciągnięcie ręki.



Naprawdę nie ma potrzeby jej przepisywania.

Na pierwszy rzut oka może się wydawać, że trzeba przepisać funkcję `log_request` w taki sposób, aby w mniejszym stopniu polegała na swoich argumentach... przy założeniu, że w ogóle jest to możliwe. Okazuje się jednak, że framework Flask zapewnia dekorator, który może nam tu pomóc.

Dekorator ten, noszący nazwę `copy_current_request_context`, zapewnia, że żądanie HTTP, które jest aktywne, gdy funkcja jest wywoływana, *pozostaje* aktywne nawet wtedy, gdy funkcja ta jest w późniejszym czasie wykonywana w wątku. Aby z niego skorzystać, musisz dodać `copy_current_request_context` do listy importowanych elementów znajdującej się na samym początku kodu Twojej aplikacji WWW.

Podobnie jak ma to miejsce w przypadku wszystkich innych dekoratorów, również te powinieneś zastosować do istniejącej funkcji, korzystając przy tym ze standardowej składni ze znakiem `@`. Istnieje tu jednak pewne zastrzeżenie. Funkcja, która ma być dekorowana, powinna być zdefiniowana *wewnątrz* funkcji, która ją wywołuje. Oznacza to, że dekorowana funkcja musi być zagnieżdżona w środku wywołującej ją funkcji (tzn. stanowić wewnętrzną funkcję).

Ćwiczenie

Poniżej znajdziesz listę zadań, które chcemy, abyś wykonał (po tym, gdy zaktualizujesz już listę rzeczy importowanych z modułu Flask).

1. Weź funkcję `log_request` i zagnieźdź ją wewnątrz funkcji `do_search`.
2. Udekoruj funkcję `log_request` za pomocą dekoratora `@copy_current_request_context`.
3. Sprawdź, czy błędy czasu wykonania, z którymi mieliśmy ostatnio do czynienia, zostały skutecznie wyeliminowane.

Ćwiczenie Rozwiązanie

Poprosiliśmy Cię o zrobienie trzech wymienionych poniżej rzeczy.

1. Weź funkcję `log_request` i zagnieźdź ją wewnątrz funkcji `do_search`.
2. Udekoruj funkcję `log_request` za pomocą dekoratora `@copy_current_request_context`.
3. Sprawdź, czy błędy czasu wykonania, z którymi mieliśmy ostatnio do czynienia, zostały skutecznie wyeliminowane.

Poniżej możesz się przekonać, jak wygląda funkcja `do_search` po tym, jak wykonaliśmy zadania 1. i 2. (zadanie 3. omówimy na następnej stronie).

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15) # To powoduje, że funkcja log_request działa naprawdę powoli...
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """insert into log
                      (phrase, letters, ip, browser_string, results)
                      values
                      (%s, %s, %s, %s, %s)"""
            cursor.execute(_SQL, (req.form['phrase'],
                                  req.form['letters'],
                                  req.remote_addr,
                                  req.user_agent.browser,
                                  res, ))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Oto Twoje wyniki:'
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logowanie się nie powiodło; wystąpił błąd:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Dekorator został zastosowany do funkcji log_request.

Funkcja log_request jest teraz zdefiniowana (zagnieżdżona) wewnątrz funkcji do_search.

Cała reszta tego kodu pozostaje bez zmian.

Nie istnieją głupie pytania

P: Czy zabezpieczanie wywołania funkcji `log_request` za pomocą konstrukcji `try-except` ma nadal sens?

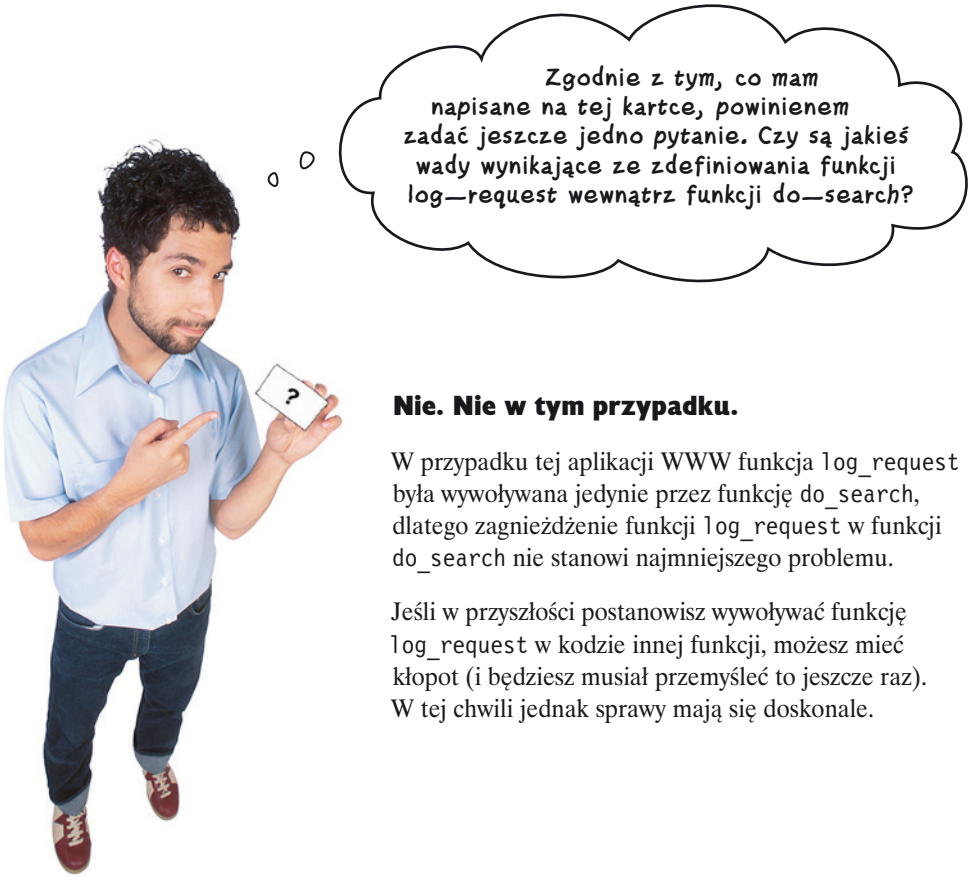
O: Nie, jeśli masz nadzieję zareagować na błąd czasu wykonania związany z funkcją `log_request`, ponieważ konstrukcja `try-except` zakończy swoje działanie, zanim swoje działanie rozpocznie wątek. Twój system może jednak mieć problem podczas próby utworzenia nowego wątku, więc uznaliśmy, że pozostawienie konstrukcji `try-except` w tej funkcji z pewnością nam tu nie zaszkodzi.

Jazda próbna

Pora na zadanie 3. Gdy weźmiemy w obroty ostatnią wersję programu *vsearch4web.py*, przekonamy się, że błędy wykonania, które pojawiły się podczas ostatniej *Jazdy próbnej*, należą już do przeszłości. Zawartość okna terminala, w którym uruchomiłeś swoją aplikację WWW, potwierdza, że wszystko jest w jak najlepszym porządku.

```
...
127.0.0.1 - - [04/Jul/2017 22:42:03] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [04/Jul/2017 22:42:07] "POST /search4 HTTP/1.1" 200 -
127.0.0.1 - - [04/Jul/2017 22:42:24] "GET /login HTTP/1.1" 200 -
127.0.0.1 - - [04/Jul/2017 22:42:33] "GET /viewlog HTTP/1.1" 200 -
127.0.0.1 - - [04/Jul/2017 22:43:15] "GET /viewlog HTTP/1.1" 200 -
```

Nie ma już żadnych strasznych wyjątków czasu wykonania. Widoczne tu dwusetki oznaczają, że z Twoją aplikacją WWW wszystko jest w porządku. A 15 sekund po tym, jak przeprowadzisz wyszukiwanie, Twoja aplikacja zaloguje w końcu do bazy danych szczegóły dotyczące żądania BEZ zmuszania użytkownika aplikacji do oczekiwania :)



Nie. Nie w tym przypadku.

W przypadku tej aplikacji WWW funkcja `log_request` była wywoływana jedynie przez funkcję `do_search`, dlatego zagnieżdżenie funkcji `log_request` w funkcji `do_search` nie stanowi najmniejszego problemu.

Jeśli w przyszłości postanowisz wywoływać funkcję `log_request` w kodzie innej funkcji, możesz mieć kłopot (i będziesz musiał przemyśleć to jeszcze raz). W tej chwili jednak sprawy mają się doskonale.

Czy Twoja aplikacja jest teraz odporna?

Poniżej znajdziesz cztery pytania postawione w rozdziale 11.

1. **Co się stanie, gdy połączenie z bazą danych zawiedzie?**
2. **Czy Twoja aplikacja jest zabezpieczona przed atakami sieciowymi?**
3. **Co się stanie, gdy coś będzie długo trwało?**
4. **Co się stanie, gdy wywołanie funkcji się nie uda?**

Twoja aplikacja obsługuje teraz szereg wyjątków czasu wykonania dzięki zastosowaniu konstrukcji `try-except` oraz kilku niestandardowym wyjątkom, które możesz zgłaszać i przechwytywać zgodnie z potrzebą.

Jeśli wiesz, że coś może pójść źle w czasie wykonania, uzbrój swój kod przeciwko wszelkim wyjątkom, które mogą się zdarzyć. Pozwala to poprawić ogólną odporność aplikacji, co jest zdecydowanie dobrą rzeczą.

Zauważ, że istnieją jeszcze inne obszary, w których można zwiększyć odporność aplikacji. Poświęciłeś mnóstwo czasu na dodawanie kodu `try-except` do funkcji `view_the_log`, która wykorzystywała możliwości zapewniane przez menedżer kontekstu `UseDatabase`. Klasa ta była używana *również* wewnątrz funkcji `log_request` i prawdopodobnie także powinna zostać odpowiednio zabezpieczona (zadanie to pozostawiamy Ci jako pracę domową).

Twoja aplikacja WWW jest bardziej responsywna dzięki zastosowaniu wątku do wykonania zadania, które musi być kiedyś w końcu przeprowadzone, lecz niekoniecznie od razu. To dobra strategia projektowa, ale musisz uważać, aby nie przesadzić z wątkami. Przykład przedstawiony w tym rozdziale jest bardzo prosty. Bardzo łatwo jednak utworzyć wykorzystujący wątki kod, którego nikt nie będzie w stanie zrozumieć i który doprowadzi Cię do szału, gdy będziesz musiał go debugować. **Wątków używaj ostrożnie.**

Jeśli chodzi o odpowiedź na pytanie nr 3 — *co się stanie, gdy coś będzie długo trwało?* — zastosowanie mechanizmu wątków poprawiło (z perspektywy użytkownika aplikacji) wydajność zapisu w bazie danych, lecz nie odczytu. Mamy tu do czynienia z przypadkiem, w którym po prostu trzeba czekać na nadejście danych po odczycie, niezależnie od tego, ile miałoby to potrwać, ponieważ nasza aplikacja WWW nie jest w stanie dalej działać bez tych danych.

Aby przyspieszyć operację odczytu (zakładając, że to właśnie ona jest w tym przypadku w największej mierze odpowiedzialna za spowolnienie), będziesz prawdopodobnie musiał rozejrzeć się za alternatywnym (szybszym) rozwiązaniem bazodanowym. Jest to jednak zmartwienie na przyszłość, którym nie będziemy się już przejmować w niniejszej książce.

Mimo to jednak w następnym i ostatnim już rozdziale w istocie będziemy zajmować się kwestią wydajności, lecz zrobimy to, omawiając temat, który wszyscy doskonale rozumieją i który przedstawiliśmy już wcześniej, a mianowicie przetwarzanie danych w pętli.

Kod przykładów z rozdziału 11³/₄ (1. z 2)

To jest najnowsza i najlepsza wersja pliku vsearch4web.py.

```
from flask import Flask, render_template, request, escape, session
from flask import copy_current_request_context
from vsearch import search4letters

from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLError
from checker import check_logged_in

from threading import Thread
from time import sleep

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
   session['logged_in'] = True
   return 'Teraz jesteś zalogowany.'

@app.route('/logout')
def do_logout() -> str:
   session.pop('logged_in')
   return 'Teraz jesteś wylogowany.'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

   @copy_current_request_context
   def log_request(req: 'flask_request', res: str) -> None:
      sleep(15) # To powoduje, że funkcja log_request działa naprawdę powoli...
      with UseDatabase(app.config['dbconfig']) as cursor:
         _SQL = """insert into log
                   (phrase, letters, ip, browser_string, results)
                   values
                   (%s, %s, %s, %s, %s)"""
         cursor.execute(_SQL, (req.form['phrase'],
                               req.form['letters'],
                               req.remote_addr,
                               req.user_agent.browser,
                               res, ))

   phrase = request.form['phrase']
   letters = request.form['letters']
   title = 'Oto Twoje wyniki:'
```

Pozostała część funkcji do_search znajduje się na początku następnej strony.

Kod przykładów z rozdziału 11 3/4 (2. z 2)

```
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logowanie się nie powiodło; wystąpił błąd:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Witamy na stronie internetowej search4letters!')

@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                      from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        # raise Exception("Jakiś nieznany wyjątek.")
        titles = ('Fraza' , 'Litery' , 'Adres klienta', 'Agent użytkownika', 'Wyniki')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
    except ConnectionError as err:
        print('Czy Twoja baza danych jest włączona? Błąd:', str(err))
    except CredentialsError as err:
        print('Problemy z identyfikatorem użytkownika lub hasłem dostępu. Błąd:', str(err))
    except SQLError as err:
        print('Czy Twoje zapytanie jest poprawne? Błąd:', str(err))
    except Exception as err:
        print('Coś poszło źle:', str(err))
    return 'Błąd'

app.secret_key = 'NigdyNieZgadnieszMojegoTajnegoKlucza'

if __name__ == '__main__':
    app.run(debug=True)
```

To jest reszta kodu funkcji do_search.

12. Zaawansowana iteracja

Szalone zapętlenie

Właśnie przyszedł mi do głowy wspaniały pomysł. Co by było, gdybym mogła przyspieszyć działanie swoich pętli?

To zdumiewające, ile czasu nasze programy spędzają w pętlach.

Nie jest to duża niespodzianka, ponieważ zadanie większości programów polega na szybkim wykonywaniu pewnych czynności bardzo wiele razy. Jeśli chodzi o optymalizację pętli, do dyspozycji mamy dwa podstawowe sposoby: pierwszy polega na poprawieniu składni pętli (aby ułatwić definiowanie pętli), a drugi sprowadza się do poprawienia sposobu działania pętli (aby były one wykonywane szybciej). Na wczesnym etapie istnienia języka Python 2 (czyli *dawno*, *dawno* temu) jego projektanci wprowadzili pewną cechę języka, która stanowiła implementację obydwu tych sposobów i otrzymała dość dziwną nazwę: **komprehencja**[1] (ang. *comprehension*). Nie daj się jednak zniechęcić tej dziwacznej nazwie, ponieważ zanim dojdziesz do końca tego rozdziału, będziesz się zastanawiał, jak do tej pory w ogóle mogłeś obyć się bez komprehencji.

[1] Słowo „komprehencja" funkcjonuje w języku polskim w zupełnie innym znaczeniu, jednak z powodu braku w rodzimej literaturze informatycznej odpowiedniego tłumaczenia (tj. prostego, powszechnie stosowanego, jednoznacznego i właściwie oddającego sens tego pojęcia) postanowiłem tu wybrać mniejsze zło i zastosować pewnego rodzaju kalkę językową, którą łatwo jest skojarzyć z oryginalnym terminem. Słowa odpowiadającego znaczeniowo samemu wyrazowi „comprehension" nie udało mi się znaleźć w innych polskich książkach o Pythonie, spotyka się je wyłącznie w połączeniach takich jak „list comprehension", które jest zwykle tłumaczone jako „lista składana" lub „wyrażenie listowe" — *przyp. tłum.*

Brzęczyki z Bahamów mają dokąd lecieć

Aby przekonać się, co możesz osiągnąć dzięki zastosowaniu komprehencji pętli, przyjrzysz się teraz pewnym „prawdziwym” danym.

Mamy dokąd lecieć, mamy z kim się spotkać...

Działająca w Nassau na wyspie New Providence firma *Brzęczyki z Bahamów* oferuje krótkodystansowe loty pomiędzy wybranymi lotniskami na większych wyspach archipelagu. Ta linia lotnicza jako pierwsza na rynku wprowadziła elastyczny rozkład lotów: na podstawie zapotrzebowania z poprzedniego dnia przewiduje ona (choć jest to tylko nieco bardziej fantazyjne określenie słowa „zgaduje”), ile lotów będzie potrzebnych danego dnia. Na końcu każdego dnia *Centrala BB* generuje na następny dzień plan lotów, który przyjmuje postać tekstowego pliku CSV (ang. *comma-separated value*, wartości rozdzielone przecinkami).

Poniżej możesz zobaczyć, co zawiera plik CSV z rozkładem lotów na jutro.

Nagłówek informuje nas, że mamy się spodziewać dwóch kolumn danych, z których jedna zawiera godziny, druga zaś — miejsca docelowe.

```
GODZINA,MIEJSCE DOCELOWE
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

To zawartość standardowego pliku CSV, którego pierwszy wiersz zawiera nagłówek. Wszystko wygląda tu świetnie poza tym, że każdy tekst zapisany jest WIELKIMI LITERAMI (co trochę trąci myszką).

Reszta pliku CSV zawiera dane dotyczące samych lotów.

Centrala nazwała ten plik CSV *buzzers.csv*.

Gdyby poproszono Cię o odczytanie danych z tego pliku i wyświetlenie ich na ekranie, skorzystałbyś z instrukcji `with`. Poniżej możesz zobaczyć, co zrobiliśmy w wierszu poleceń środowiska IDLE po użyciu modułu `os` języka Python w celu przejścia do katalogu zawierającego nasz plik.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> import os
>>> os.chdir('/Users/lsuma/Desktop/Python/buzzdata')
>>>
>>> with open('buzzers.csv') as raw_data:
        print(raw_data.read())

GODZINA,MIEJSCE DOCELOWE
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND

>>> |
```

Ln: 20 Col: 4

Tutaj umieść ścieżkę dostępu do używanego przez siebie folderu.

Metoda read zasysa za jednym zamachem wszystkie znaki znajdujące się w pliku.

To są surowe dane CSV pochodzące z pliku.

Porady maniaka

Więcej informacji na temat formatu CSV możesz znaleźć pod adresem *https://pl.wikipedia.org/wiki/CSV_(format_pliku)*.

Odczytywanie danych CSV jako list

Dane CSV w swojej surowej formie nie są szczególnie użyteczne. Byłyby znacznie bardziej przydatne, gdyby dało się je odczytać i podzielić każdy wiersz w miejscu występowania przecinka, upraszczając w ten sposób dostęp do poszczególnych danych składowych.

Choć da się dokonać tego podziału za pomocą własnego kodu Pythona (przy użyciu metody `split` należącej do obiektu reprezentującego łańcuch znakowy), korzystanie z danych CSV jest tak powszechną czynnością, że *biblioteka standardowa* zapewnia moduł o nazwie `csv`, który z pewnością może się nam tu przydać.

Poniżej znajdziesz kolejną niewielką pętlę `for`, która prezentuje możliwości oferowane przez moduł `csv`. W przeciwieństwie do poprzedniego przykładu, w którym korzystałeś z metody `read` w celu pobrania całej zawartości pliku *za jednym zamachem*, w kodzie przedstawionym poniżej użyliśmy metody `csv.reader`, aby odczytać plik CSV *po jednym wierszu* naraz w pętli `for`. W każdej iteracji pętla ta przypisuje każdy wiersz danych CSV do zmiennej (o nazwie `line`), która jest następnie wyświetlana na ekranie.

Otwórz plik za pomocą instrukcji with...

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> import csv
>>>
>>> with open('buzzers.csv') as data:
        for line in csv.reader(data):
                print(line)


['GODZINA', 'MIEJSCE DOCELOWE']
['09:35', 'FREEPORT']
['17:00', 'FREEPORT']
['09:55', 'WEST END']
['19:00', 'WEST END']
['10:45', 'TREASURE CAY']
['12:00', 'TREASURE CAY']
['11:45', 'ROCK SOUND']
['17:55', 'ROCK SOUND']
>>>
>>>
                                                Ln: 40  Col: 4
```

To wygląda już lepiej: każdy wiersz danych z pliku CSV został przekształcony w listę.

...następnie odczytaj dane wiersz po wierszu, korzystając z metody csv.reader.

Moduł `csv` wykonuje tu kawał dobrej roboty. Każdy wiersz surowych danych jest odczytywany z pliku, a następnie „magicznie” przekształcany w dwuelementową listę.

Oprócz tego, że informacje tworzące nagłówek (znajdujące się w pierwszym wierszu pliku) zostały zwrócone w postaci listy, formę oddzielnej listy przybrała też każda para składająca się z godziny odlotu i miejsca docelowego. Zauważ, jaki *typ* mają poszczególne elementy danych, które zostały w ten sposób zwrócone: wszystko jest tutaj łańcuchem znakowym, mimo że pierwszy element każdej listy (ewidentnie) reprezentuje godzinę.

Moduł `csv` ma jeszcze kilka asów w rękawie. Inną interesującą metodą jest `csv.DictReader`. Przekonajmy się, co może ona dla nas zrobić.

Odczytywanie danych CSV jako słowników

Poniżej znajdziesz kod podobny do poprzedniego przykładu, jedyną różnicą jest to, że zamiast metody `csv.reader` korzystamy tu z metody `csv.DictReader`. Gdy używana jest ta ostatnia, dane z pliku CSV są zwracane w postaci kolekcji słowników, których klucze tworzone są na podstawie wiersza nagłówka tego pliku, a wartości pobierane są z każdego z jego kolejnych wierszy.

Zastosowanie metody csv.DictReader to prosta zmiana, lecz robi wielką różnicę. To, co (poprzednim razem) było wierszami list, teraz jest wierszami słowników.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>>
>>> with open('buzzers.csv') as data:
        for line in csv.DictReader(data):
            print(line)

OrderedDict([('GODZINA', '09:35'), ('MIEJSCE DOCELOWE', 'FREEPORT')])
OrderedDict([('GODZINA', '17:00'), ('MIEJSCE DOCELOWE', 'FREEPORT')])
OrderedDict([('GODZINA', '09:55'), ('MIEJSCE DOCELOWE', 'WEST END')])
OrderedDict([('GODZINA', '19:00'), ('MIEJSCE DOCELOWE', 'WEST END')])
OrderedDict([('GODZINA', '10:45'), ('MIEJSCE DOCELOWE', 'TREASURE CAY')])
OrderedDict([('GODZINA', '12:00'), ('MIEJSCE DOCELOWE', 'TREASURE CAY')])
OrderedDict([('GODZINA', '11:45'), ('MIEJSCE DOCELOWE', 'ROCK SOUND')])
OrderedDict([('GODZINA', '17:55'), ('MIEJSCE DOCELOWE', 'ROCK SOUND')])
>>>
>>>
```

Ln: 55 Col: 4

Klucze

Wartości

```
GODZINA,MIEJSCE DOCELOWE
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

Przypomnij sobie, że surowe dane w pliku miały taką postać.

Nie ma wątpliwości, że to potężne narzędzie — za pomocą jednego wywołania metody `DictReader` moduł `csv` przetworzył surowe dane zapisane w pliku CSV w kolekcję słowników Pythona.

Wyobraź sobie jednak, że otrzymałeś zadanie polegające na skonwertowaniu surowych danych zapisanych w pliku CSV zgodnie z następującymi wymaganiami:

1. **Skonwertuj godziny odlotów z formatu 24-godzinnego na format 12-godzinny (AM/PM).**
2. **Skonwertuj zapis miejsc docelowych z WIELKICH LITER na postać nazw własnych.**

Zadania te ze swej natury nie należą do szczególnie trudnych. Jeśli jednak potraktujesz surowe dane jako kolekcję list lub kolekcję słowników, zadania te mogą się skomplikować. Napiszmy zatem własną pętlę `for`, aby wprowadzać odczytywane dane do pojedynczego słownika, którego będzie można potem użyć do przeprowadzenia konwersji w znacznie prostszy sposób.

Cofnijmy się troszeczkę

Zamiast używać metody `csv.reader` lub `csv.DictReader`, sprawmy, aby nasz kod konwertował surowe dane zapisane w pliku CSV na *pojedynczy* słownik, którym będziemy się mogli posłużyć w celu przeprowadzenia wymaganych przekształceń.

Ucięliśmy sobie pogawędkę z ludźmi z *Centrali* firmy *Brzęczyki z Bahamów* i powiedzieli nam oni, że są bardzo zadowoleni z konwersji, którą mamy zamiar przeprowadzić, ale nadal chcieliby przechowywać dane w „surowej formie", ponieważ właśnie w takiej postaci spodziewają się je otrzymać ich przestarzałe tablice odlotów; godziny odlotów powinny mieć format 24-godzinny, a miejsca docelowe powinny być zapisane wyłącznie za pomocą WIELKICH LITER.

Surowe dane

```
GODZINA,MIEJSCE DOCELOWE
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

Czy da się podzielić każdy wiersz na dwie części, wykorzystując przecinek w roli separatora?

Konwersję mógłbyś przeprowadzić bezpośrednio na surowych danych przechowywanych w Twoim pojedynczym słowniku, lepiej będzie jednak, gdy zapewnisz, aby odbywała się ona na *kopii* danych, nie zaś na odczytanych oryginalnych wierszach. Choć obecnie nie jest to jeszcze całkiem jasne, plotki docierające do nas z *Centrali* wydają się wskazywać, że kod, który opracujesz, będzie prawdopodobnie musiał współdziałać z jakimiś istniejącymi już systemami. Zamiast więc stawać w obliczu perspektywy przekształcania danych z powrotem w ich wyjściową, surową formę, zapisujmy je w pojedynczym słowniku w takiej postaci, jaką obecnie mają, a następnie konwertujmy odpowiednie kopie, gdy tylko będzie to potrzebne (pozostawiając *nietknięte* surowe dane w oryginalnym słowniku).

Odczytywanie surowych danych i umieszczanie ich w słowniku nie wiąże się z jakimś niewyobrażalnie wielkim nakładem pracy (jak widać to w zamieszczonych powyżej i poniżej przykładach kodu wykorzystującego moduł `csv`). We fragmencie kodu pokazanym poniżej otwierany jest plik, a jego pierwszy wiersz jest odczytywany i ignorowany (ponieważ nie są nam do niczego potrzebne informacje zawarte w nagłówku). Następnie za pomocą pętli `for` odczytujemy każdy wiersz surowych danych, dzieląc go na dwie części w miejscu wskazywanym przez przecinek, przy czym godzina odlotu jest używana jako *klucz* słownika, a miejsce docelowe odgrywa rolę jego *wartości*.

Otwórz plik, tak jak robiłeś to wcześniej.

Utwórz nowy, pusty słownik o nazwie „flights".

Przetwórz każdą linię

Zignoruj te informacje z nagłówka.

Podziel wiersz danych w miejscu występowania przecinka, co spowoduje zwrócenie dwóch danych: klucza (godziny odlotu) oraz wartości (miejsca docelowego).

Przypisz wartość miejsca docelowego do klucza będącego godziną odlotu.

Wyświetl zawartość tego słownika, która wygląda dość nieporządnie aż do czasu, gdy...

...za pomocą biblioteki pretty-printing wygenerujemy dane wyjściowe nieco milsze dla oka.

Uwzględnienie znaku nowego wiersza daje trochę dziwny efekt, nie sądzisz?

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> with open('buzzers.csv') as data:
        ignore = data.readline()
        flights = {}
        for line in data:
            k, v = line.split(',')
            flights[k] = v
>>> flights
{'09:35': 'FREEPORT\n', '17:00': 'FREEPORT\n', '09:55': 'WEST END\
n', '19:00': 'WEST END\n', '10:45': 'TREASURE CAY\n', '12:00': 'TR
EASURE CAY\n', '11:45': 'ROCK SOUND\n', '17:55': 'ROCK SOUND\n'}
>>>
>>> import pprint
>>> pprint.pprint(flights)
{'09:35': 'FREEPORT\n',
 '09:55': 'WEST END\n',
 '10:45': 'TREASURE CAY\n',
 '11:45': 'ROCK SOUND\n',
 '12:00': 'TREASURE CAY\n',
 '17:00': 'FREEPORT\n',
 '17:55': 'ROCK SOUND\n',
 '19:00': 'WEST END\n'}
>>>
```

Ln: 76 Col: 4

„Obieranie", a następnie podział Twoich surowych danych

W ostatniej instrukcji `with` została wykorzystana metoda `split` (dostępna w przypadku wszystkich obiektów łańcuchów znakowych) w celu podziału wiersza surowych danych na dwie części. W wyniku tego zwracana jest lista łańcuchów znakowych, które są (każdy z osobna) przypisywane do zmiennych `k` oraz `v`. To (przedstawione poniżej) przypisanie do wielu zmiennych naraz jest możliwe dzięki temu, że z jego lewej strony znajduje się krotka zmiennych, a kod znajdujący się z prawej strony operatora zwraca listę wartości (pamiętaj, że krotki są *niemodyfikowalnymi* listami).

```
...
k, v = line.split(',')
...
```

Krotka zmiennych znajduje się z lewej strony.

Kod, który generuje listę wartości z prawej.

Kolejna metoda, o nazwie `strip`, dostępna w przypadku obiektów łańcuchów znakowych powoduje usunięcie białych znaków z początku i końca istniejącego łańcucha. Skorzystajmy z niej, aby *przed* przeprowadzeniem operacji podziału spośród surowych danych pozbyć się niechcianych znaków nowego wiersza występujących na końcu wierszy.

Porady maniaka

Białe znaki — następujące znaki są uznawane za białe znaki w łańcuchach znakowych: *spacja*, `\t`, `\n` oraz `\r`.

Poniżej znajdziesz ostateczną wersję naszego kodu odczytującego dane. Tworzymy tu słownik o nazwie `flights`, w którym w roli kluczy używane są godziny odlotów, a miejsca docelowe (pozbawione znaku końca wiersza) odgrywają rolę wartości.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> with open('buzzers.csv') as data:
        ignore = data.readline()
        flights = {}
        for line in data:
                k, v = line.strip().split(',')
                flights[k] = v


>>>
>>> pprint.pprint(flights)
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}
>>>
```

Ln: 95 Col: 4

Ten kod „obiera" wiersz, a następnie dzieli go, aby zwrócić dane w wymaganym formacie

Mogłeś tego nie zauważyć, ale kolejność wierszy w słowniku jest inna niż układ danych w pliku. Dzieje się tak, ponieważ słowniki NIE zachowują kolejności wstawiania danych. Nie przejmuj się tym teraz.

```
GODZINA,MIEJSCE DOCELOWE
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

A co by się stało, gdybyś w swoim kodzie zmienił kolejność wywołania metod, tak jak przedstawiono poniżej?

```
line.split(',').strip()
```

Gdy wywołania metod łączysz w łańcuch w taki sposób, jak zostało to pokazane tutaj, mamy do czynienia z „łańcuchem metod".

Jak sądzisz, co się teraz wydarzy?

Zachowaj ostrożność, łącząc wywołania metod w łańcuch

Niektórzy programiści nie są zachwyceni faktem, że język Python umożliwia łączenie wywołań metod w łańcuchy (jak ma to miejsce w przypadku metod `strip` i `split` przedstawionych na poprzedniej stronie), ponieważ łańcuchy tego rodzaju mogą być mało czytelne, gdy widzi się je po raz pierwszy. Zastosowanie tej techniki jest jednak bardzo popularne wśród programistów Pythona, dlatego z pewnością nieraz zdarzy Ci się z nią zetknąć, gdy będziesz korzystał z kodu napisanego przez kogoś innego. Należy tu zachować szczególną ostrożność, ponieważ kolejność wywołań metod *nie* może być bezkarnie zmieniana.

Przykład tego, co może pójść źle, został przedstawiony poniżej. Przyjrzyj się temu kodowi (który jest bardzo podobny do tego, który został przedstawiony wcześniej). Podczas gdy w poprzednim przykładzie najpierw występowała metoda `strip`, a potem `split`, w tym kodzie najpierw wywoływana jest metoda `split`, a następnie `strip`. Zobacz, co w takiej sytuacji się dzieje.

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>>
>>> with open('buzzers.csv') as data:
        ignore = data.readline()
        flights = {}
        for line in data:
                k, v = line.split(',').strip()
                flights[k] = v


Traceback (most recent call last):
  File "<pyshell#35>", line 5, in <module>
    k, v = line.split(',').strip()
AttributeError: 'list' object has no attribute 'strip'
>>>
>>>
                                                Ln: 110  Col: 4
```

Kolejność metod połączonych w łańcuch została tutaj odwrócona względem tego, czego używałeś wcześniej.

Interpreter nie wydaje się zachwycony i mamy do czynienia z awarią; zgłoszonym błędem jest AttributeError.

Aby zrozumieć, co się tu dzieje, zastanów się nad *typem* danych znajdujących się z prawej strony operatora przypisania w trakcie wykonania łańcucha metod pokazanego powyżej.

Zanim cokolwiek się stanie, zmienna `line` jest łańcuchem znakowym. Wywołanie metody `split` na rzecz łańcucha znakowego powoduje zwrócenie listy łańcuchów, przy czym jako separator używany jest argument przekazany tej metodzie. To, co na początku było *łańcuchem znakowym* (zmienną `line`), zostało dynamicznie przekształcone w *listę*, na rzecz której została następnie wywołana kolejna metoda. W tym przykładzie była nią metoda `strip`, która powinna być wywoływana na rzecz łańcucha znakowego, *nie* zaś na rzecz listy, w związku z czym interpreter zgłasza wyjątek `AttributeError`, jako że listy nie zapewniają metody o nazwie `strip`.

Łańcuch metod przedstawiony na poprzedniej stronie nie cierpi na tę przypadłość, o czym można się przekonać poniżej.

```
...
line.strip().split(',')
...
```

W przypadku tego kodu interpreter zaczyna od przetwarzania łańcucha znakowego (przechowywanego w zmiennej `line`), z którego za pomocą metody `strip` usuwane są znajdujące się na początku lub końcu białe znaki, co daje w wyniku kolejny łańcuch znakowy, który jest następnie dzielony na listę łańcuchów znakowych przy użyciu separatora będącego przecinkiem. Nie występuje tu wyjątek `AttributeError`, ponieważ łańcuch metod nie narusza żadnych reguł typowania.

Przekształcanie danych na format, którego potrzebujesz

Mając dane zapisane w słowniku `flights`, rozważmy operacje, o których wykonanie poprosiła Cię *Centrala BB*.

Pierwsza z nich polega na przeprowadzeniu dwóch konwersji wskazanych we wcześniejszej części tego rozdziału, z czym wiąże się utworzenie po drodze nowego słownika.

1. **Skonwertuj godziny odlotów z formatu 24-godzinnego na format 12-godzinny (AM/PM).**
2. **Skonwertuj zapis miejsc docelowych z WIELKICH LITER na postać nazw własnych.**

Zastosowanie tych dwóch przekształceń względem słownika `flights` umożliwia Ci transformację słownika zaprezentowanego poniżej po lewej stronie w ten pokazany po stronie prawej.

```
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}
```

Godziny odlotów zostały skonwertowane z formatu 24-godzinnego na format 12-godzinny.

```
{'09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'
 '05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End' }
```

Nazwy miejsc docelowych zostały skonwertowane z WIELKICH LITER na postać standardową.

Zwróć uwagę, że dane przechowywane w obydwu słownikach mają to samo znaczenie, zmieniła się tylko ich reprezentacja. *Centrala* potrzebuje drugiego słownika (czyli tego, który przechowuje przekształcone dane), ponieważ jej pracownicy uważają zapisane w nim dane za bardziej uniwersalne i zrozumiałe, jak również bardziej przyjazne dla oka; sądzą oni również, że użycie WIELKICH LITER za bardzo przypomina krzyk.

W tej chwili dane znajdujące się w obydwu słownikach uporządkowane są w taki sposób, że każdy wiersz zawiera jedną kombinację godziny odlotu i miejsca docelowego. Choć pracownicy *Centrali* ucieszą się, gdy przekształcisz słownik widoczny z lewej strony w ten pokazany z prawej, zasugerowali również, że byłby on naprawdę użyteczny, gdyby dane były prezentowane tak, żeby miejsca docelowe były kluczami, a listy godzin odlotów stanowiły wartości, a więc gdyby każde miejsce docelowe wymagało użycia tylko jednego wiersza. Zobaczmy, jak będzie wyglądał *ten* słownik, zanim zabierzemy się za opracowywanie kodu odpowiedzialnego za niezbędne operacje.

Przekształcanie do postaci słownika list

Gdy dane zapisane w słowniku `flights` zostały już odpowiednio przekształcone, *Centrala* chce, abyś przeprowadził drugą z operacji (opisaną na końcu poprzedniej strony), tak jak zostało to pokazane poniżej.

To, co było wartościami, staje się kluczami.

```
{'09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'
 '05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End' }
```

```
{'Freeport': ['09:35AM', '05:00PM'],
 'Rock Sound': ['11:45AM', '05:55PM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['09:55AM', '07:00PM']}
```

To, co było kluczami, staje się listami wartości.

Pomyśl o niezbędnych tu roszadach z danymi...

Aby przekształcić surowe dane zapisane w pliku CSV do postaci słownika list przedstawionego po prawej stronie powyżej, trzeba wykonać trochę pracy. Poświęć chwilę na zastanowienie się nad tym, jak spróbowałbyś to zrobić, korzystając z możliwości języka Python, które udało Ci się już poznać.

Jeśli masz tak jak większość programistów Pythona, nie zajmie Ci dużo czasu dojście do wniosku, że pomoże Ci w tym pętla `for`. Jako główny mechanizm do powtarzania wykonania pewnych operacji pętla ta umożliwiła Ci już wydobycie danych z pliku CSV oraz wypełnienie nimi słownika `flights`, co pokazano poniżej.

To jest klasyczne zastosowanie pętli for oraz wielce popularny w języku Python idiom programistyczny.

```
with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v
```

Aż kusi, aby zasugerować tu taką zmianę pokazanego powyżej kodu, aby przeprowadzał on niezbędne transformacje surowych danych już w trakcie odczytywania ich z pliku CSV, a więc jeszcze *przed* dodaniem wierszy danych do słownika `flights`. Przypomnij sobie jednak sformułowane przez *Centralę* wymaganie, aby surowe dane pozostały nietknięte w tym słowniku — wszelkie przekształcenia mają być dokonywane na **kopiach** tych danych. Trochę komplikuje nam to sprawę, ale tylko trochę.

Wykonajmy proste konwersje

W tej chwili słownik `flights` zawiera klucze będące godzinami odlotów w formacie 24-godzinnym oraz wartości, którymi są zapisane WIELKIMI LITERAMI łańcuchy znaków reprezentujące miejsca docelowe. Powinieneś przeprowadzić dwie wstępne konwersje:

1. **Skonwertuj godziny odlotów z formatu 24-godzinnego na format 12-godzinny (AM/PM).**
2. **Skonwertuj zapis miejsc docelowych z WIELKICH LITER na postać nazw własnych.**

Druga konwersja jest bardzo prosta, dlatego zajmijmy się nią najpierw. Gdy dane mają postać łańcucha znakowego, wystarczy wywołać na ich rzecz metodę `title` zapewnianą przez obiekt tekstowy, jak zostało to zademonstrowane w poniższej sesji środowiska IDLE.

```
>>> s = "NIE MIAŁEM ZAMIARU KRZYCZEĆ."
>>> print(s)
NIE MIAŁEM ZAMIARU KRZYCZEĆ.
>>> t = s.title()
>>> print(t)
Nie Miałem Zamiaru Krzyczeć.
```

Metoda title zwraca zmodyfikowaną kopię danych przechowywanych w zmiennej s.

Ten łańcuch wygląda nieco lepiej niż poprzedni.

Pierwsza z wymienionych konwersji wymaga od nas nieco więcej pracy.

Jeśli zastanowisz się nad tym przez chwilkę, stanie się dla Ciebie jasne, że konwersja wartości `19:00` na `7:00PM` nie jest operacją banalnie prostą. Jest tak jednak tylko wówczas, gdy traktujesz wartości takie jak `19:00` jako łańcuchy znakowe. Będziesz musiał napisać mnóstwo kodu, aby przeprowadzić tego rodzaju konwersję.

Jeśli natomiast przyjmiesz, że wartość `19:00` określa czas, będziesz mógł skorzystać z możliwości oferowanych przez moduł `datetime`, który wchodzi w skład *biblioteki standardowej* języka Python. Należąca do tego modułu klasa `datetime` zapewnia możliwość konwersji łańcucha znakowego (takiego jak `19:00`) na odpowiadającą mu znaczeniowo wartość w formacie 12-godzinnym (AM/PM). Możliwość ta realizowana jest przez dwie metody oraz konstrukcje znane jako specyfikatory *formatu łańcucha znakowego* (ang. *string format specifiers*). Oto niewielka funkcja o nazwie `convert2ampm`, w której wykorzystywane są możliwości oferowane przez moduł `datetime` w celu przeprowadzenia wymaganych konwersji.

Więcej informacji na temat specyfikatorów formatu łańcuchów znakowych znajdziesz pod adresem *https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior.*

Kod gotowy do użycia

```
from datetime import datetime

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')
```

Ten łańcuch metod umożliwia przekształcenie podanej mu (w formie łańcucha znakowego) wartości czasu w formacie 24-godzinnym w łańcuch znakowy zawierający wartość czasu w formacie 12-godzinnym.

Zaostrz ołówek

Zaprzęgnijmy do pracy techniki przedstawione na poprzedniej stronie.

Poniżej znajdziesz kod odczytujący surowe dane z pliku CSV oraz sukcesywnie wypełniający nimi słownik `flights`. Zaprezentowana została tu również funkcja `convert2ampm`.

Twoim zadaniem jest napisanie pętli `for`, która pobiera dane ze słownika `flights` i przekształca należące do niego klucze na format 12-godzinny, a wartości zapisuje jako *nazwy własne*. Do przechowywania przekształconych danych ma być używany nowy słownik o nazwie `flights2`, który należy utworzyć. Skorzystaj z ołówka w celu napisania w pustych miejscach kodu tworzącego tę pętlę `for`.

Wskazówka: pamiętaj, że przy przetwarzaniu słownika za pomocą pętli `for` metoda `items` podczas każdej iteracji zwraca klucz i wartość (jako krotkę) dla każdego wiersza.

Zdefiniuj funkcję konwertującą.

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}

.....

.....

pprint.pprint(flights2)
```

Pobierz dane z pliku.

Wyświetl ładnie dane ze słownika flights przed przeprowadzeniem konwersji.

Nowy słownik o nazwie „flights2" zostaje utworzony jako pusty słownik.

Tutaj dodaj swoją pętlę for.

Wyświetl ładnie dane ze słownika flights2 w celu potwierdzenia, że konwersje działają zgodnie z oczekiwaniami.

Zaostrz ołówek
Rozwiązanie

Twoim zadaniem było napisanie pętli `for`, która pobiera dane ze słownika `flights` i przekształca należące do niego klucze na format 12-godzinny, a wartości zapisuje jako *nazwy własne*. Do przechowywania przekształconych danych miał być używany nowy słownik o nazwie `flights2`, który należało utworzyć. Miałeś skorzystać z ołówka w celu napisania w pustych miejscach kodu tworzącego tę pętlę `for`.

Cały ten kod zapisaliśmy w pliku o nazwie „do_convert.py".

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()

pprint.pprint(flights2)
```

Metoda items zwraca każdy wiersz znajdujący się w słowniku flights.

W każdej iteracji klucz (znajdujący się w zmiennej k) jest konwertowany na format 12-godzinny, a następnie używany jako klucz w nowym słowniku.

Wartość (znajdująca się w zmiennej v) jest konwertowana na format nazwy własnej, a następnie przypisywana do skonwertowanego klucza.

Jazda próbna

Gdy wykonasz przedstawiony powyżej program, na ekranie zostanie wyświetlona zawartość dwóch słowników (są one też pokazane poniżej, jeden obok drugiego). Konwersja działa, chociaż kolejność elementów w słownikach jest różna, ponieważ interpreter **nie** zachowuje *kolejności wstawiania* danych przy wypełnianiu nowego słownika.

To jest zawartość słownika flights.

```
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}
```

```
{'05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End',
 '09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'}
```

To jest zawartość słownika flights2.

Surowe dane zostały przekształcone.

Czy zauważyłeś pewien wzorzec w swoim kodzie?

Przyjrzyj się raz jeszcze programowi, który właśnie uruchomiłeś. W kodzie tym *dwukrotnie* występuje pewien bardzo powszechnie spotykany wzorzec programowania. Czy jesteś w stanie go dostrzec?

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
   return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
   ignore = data.readline()
   flights = {}
   for line in data:
      k, v = line.strip().split(',')
      flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}
for k, v in flights.items():
   flights2[convert2ampm(k)] = v.title()

pprint.pprint(flights2)
```

Jeśli odpowiedziałeś: „pętla `for`”, masz rację, ale to dopiero pół prawdy. Pętla `for` stanowi część wspomnianego wzorca, ale przyjrzyj się jeszcze raz kodowi, który ją *otacza*. Czy dostrzegłeś coś innego?

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
   return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
   ignore = data.readline()
   flights = {}
   for line in data:
      k, v = line.strip().split(',')
      flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}
for k, v in flights.items():
   flights2[convert2ampm(k)] = v.title()

pprint.pprint(flights2)
```

Każdą pętlę for poprzedza operacja tworzenia nowej, pustej struktury danych (w tym przypadku jest to słownik).

Każdy zestaw pętli for zawiera kod wykonujący operację dodawania danych do nowej struktury danych, które to dane powstały w wyniku przetwarzania pewnych istniejących danych.

Dostrzeganie wzorców w przypadku list

Przykłady przedstawione na poprzedniej stronie pokazały wzorzec programistyczny występujący w przypadku słowników. Polega on na rozpoczynaniu od nowego, pustego słownika, a następnie użyciu pętli `for` w celu przetworzenia istniejącego słownika i sukcesywnego generowania danych zapisywanych w nowym słowniku.

```
flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()
```

To jest nowy, początkowo pusty słownik.

To jest istniejący słownik.

Zwykła pętla for przetwarza istniejące dane.

Istniejące dane są używane do generowania kluczy i wartości, które są wstawiane do nowego słownika.

Wzorzec ten pojawia się także w przypadku list, gdzie łatwiej go dostrzec. Przyjrzyj się pokazanej poniżej sesji IDLE, w której klucze (czyli godziny odlotów) oraz wartości (czyli miejsca docelowe) są wydobywane ze słownika `flights` jako listy, a następnie są konwertowane na nowe listy przy użyciu przedstawionego poniżej wzorca programistycznego (opisanego za pomocą komentarzy ponumerowanych od 1. do 4.).

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>>
>>> flight_times = []
>>> for ft in flights.keys():
        flight_times.append(convert2ampm(ft))

>>> print(flight_times)
['09:35AM', '05:00PM', '09:55AM', '07:00PM', '10:45AM', '12:00PM'
, '11:45AM', '05:55PM']
>>>
>>> destinations = []
>>> for dest in flights.values():
        destinations.append(dest.title())

>>> print(destinations)
['Freeport', 'Freeport', 'West End', 'West End', 'Treasure Cay',
'Treasure Cay', 'Rock Sound', 'Rock Sound']
>>>
>>>
```

Ln: 88 Col: 4

1. Zacznij od nowej, pustej listy.

2. Iteruj po wszystkich godzinach odlotu.

3. Dołącz skonwertowane dane do nowej listy.

4. Wyświetl dane znajdujące się na nowej liście.

1. Zacznij od nowej, pustej listy.

2. Iteruj po wszystkich miejscach docelowych.

3. Dołącz skonwertowane dane do nowej listy.

4. Wyświetl dane znajdujące się na nowej liście.

Ten wzorzec jest wykorzystywany tak często, że twórcy języka Python postanowili zapewnić wygodną notację umożliwiającą skrócenie zapisu odpowiedniej konstrukcji. Notacja ta nosi nazwę **komprehencji**. Przekonajmy się, z czym wiąże się tworzenie tej konstrukcji.

Przekształcanie wzorców w komprehencje

Jako przykład weźmy najnowszą wersję pętli `for`, która przetwarzała miejsca docelowe. Poniżej możesz ją zobaczyć po raz kolejny.

```
destinations = []
for dest in flights.values():
    destinations.append(dest.title())
```

1. Zacznij od nowej, pustej listy.

2. Iteruj po wszystkich miejscach docelowych.

3. Dołącz skonwertowane dane do nowej listy.

Wbudowany w język Python mechanizm **komprehencji** umożliwia Ci zapisanie przedstawionych powyżej trzech wierszy kodu za pomocą jednego wiersza.

Aby pokazać Ci sposób tworzenia tego rodzaju konstrukcji, przekształcimy te trzy wiersze w komprehencję, wykonując cały proces krok po kroku.

Zacznij od utworzenia nowej, pustej listy, którą przypiszesz do nowej zmiennej (nazwanej w naszym przykładzie `more_dests`).

```
more_dests = []
```

1. Zacznij od nowej, pustej listy (i nadaj jej nazwę).

Określ sposób, w jaki istniejące dane (w tym przykładzie zapisane w słowniku `flights`) mają być iterowane, korzystając przy tym z dobrze znanej Ci notacji pętli `for`. Odpowiedni kod umieść w nawiasach kwadratowych nowej listy (zwróć tu uwagę na *brak* dwukropka na końcu kodu pętli `for`).

```
more_dests = [for dest in flights.values()]
```

2. Iteruj po wszystkich miejscach docelowych.

Zauważ, że NIE ma tu dwukropka.

Aby ukończyć komprehencję, określ przekształcenie, które ma zostać zastosowane do danych (zapisanych w zmiennej `dest`) i umieść to przekształcenie *przed* słowem kluczowym `for` (zwróć tu uwagę na *brak* wywołania metody `append`, które zapewnia komprehencja).

```
more_dests = [dest.title() for dest in flights.values()]
```

3. Dołącz skonwertowane dane do nowej listy bez jawnego wywołania metody append.

I to już wszystko. Widoczny tu pojedynczy wiersz kodu odpowiada funkcjonalnie trzem wierszom kodu przedstawionym na początku tej strony. Śmiało, uruchom ten wiersz kodu w swoim wierszu poleceń powłoki Pythona, aby przekonać się, że lista `more_dests` zawiera te same dane co lista `destinations`.

Przyjrzyj się uważnie swojej komprehencji

Przyjrzyjmy się nieco uważniej naszej komprehencji. Poniżej znajdziesz zarówno oryginalne trzy wiersze kodu, jak i mieszczącą się w jednym wierszu komprehencję, która wykonuje to samo zadanie.

Pamiętaj, że obydwie wersje tworzą nowe listy (`destinations` oraz `more_dests`) zawierające dokładnie te same dane.

```
destinations = []
for dest in flights.values():
    destinations.append(dest.title())
```

```
more_dests = [dest.title() for dest in flights.values()]
```

Możliwe jest również wybranie części składowych oryginalnych trzech wierszy kodu i wskazanie miejsc w komprehencji, w których zostały one użyte.

```
destinations = []
for dest in flights.values():
    destinations.append(dest.title())

more_dests = [dest.title() for dest in flights.values()]
```

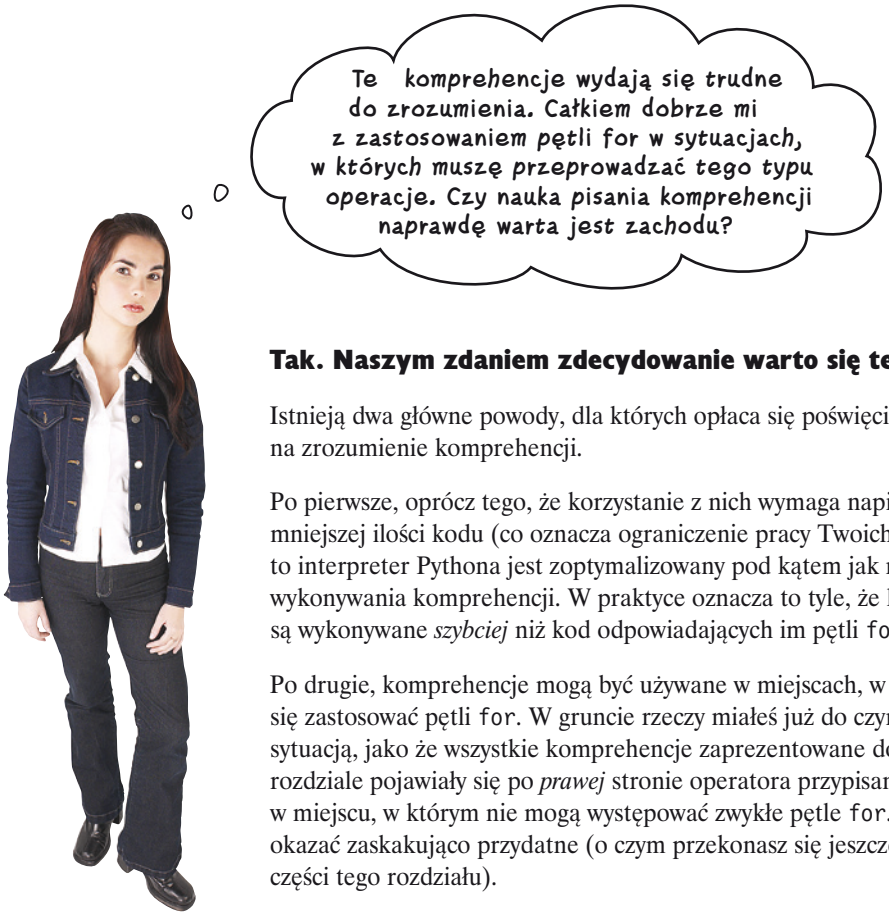
Jeśli dostrzeżesz ten wzorzec w innym fragmencie kodu, będziesz mógł łatwo przekształcić go w komprehencję. Poniżej został na przykład zaprezentowany kod (odpowiedzialny za generowanie listy godzin odlotu w formacie 12-godzinnym) pochodzący z wcześniejszej części rozdziału, który to kod został przekształcony w kolejną komprehencję.

```
flight_times = []
for ft in flights.keys():
    flight_times.append(convert2ampm(ft))
```

Te dwa fragmenty kodu robią TO SAMO.

```
fts2 = [convert2ampm(ft) for ft in flights.keys()]
```

O co tyle szumu?



Tak. Naszym zdaniem zdecydowanie warto się tego nauczyć.

Istnieją dwa główne powody, dla których opłaca się poświęcić trochę czasu na zrozumienie komprehencji.

Po pierwsze, oprócz tego, że korzystanie z nich wymaga napisania mniejszej ilości kodu (co oznacza ograniczenie pracy Twoich palców), to interpreter Pythona jest zoptymalizowany pod kątem jak najszybszego wykonywania komprehencji. W praktyce oznacza to tyle, że komprehencje są wykonywane *szybciej* niż kod odpowiadających im pętli `for`.

Po drugie, komprehencje mogą być używane w miejscach, w których nie da się zastosować pętli `for`. W gruncie rzeczy miałeś już do czynienia z taką sytuacją, jako że wszystkie komprehencje zaprezentowane dotąd w tym rozdziale pojawiały się po *prawej* stronie operatora przypisania, a więc w miejscu, w którym nie mogą występować zwykłe pętle `for`. Może się to okazać zaskakująco przydatne (o czym przekonasz się jeszcze w dalszej części tego rozdziału).

Zastosowanie komprehencji nie ogranicza się wyłącznie do list

Komprehencje, które miałeś okazję zobaczyć do tej pory, tworzyły nowe listy, dlatego każda z nich określana jest mianem **komprehencji listy** (lub też w skrócie *listcomp*). Jeśli Twoja komprehencja tworzy nowy słownik, jest znana jako **komprehencja słownika** (*dictcomp*). A żeby nie pominąć żadnej struktury danych, możesz również zdefiniować **komprehencję zbioru** (*setcomp*).

Nie istnieje nic takiego jak *komprehencja krotki*; w dalszej części tego rozdziału wyjaśnimy, dlaczego tak jest.

Najpierw jednak przyjrzyjmy się komprehencji słownika.

Definiowanie komprehencji słownika

Przypomnij sobie przedstawiony wcześniej w tym rozdziale kod, który odczytywał dane z pliku CSV i umieszczał je w słowniku o nazwie `flights`. Dane te były następnie przekształcane w nowy słownik o nazwie `flights2`, w którym kluczami są godziny odlotów zapisane w formacie 12-godzinnym, a wartościami — miejsca docelowe zapisane jak nazwy własne.

```
...
flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()
...
```

Ten kod jest zgodny ze „wzorcem komprehencji".

Zmieńmy te trzy wiersze kodu w komprehencję słownika.

Zacznij od przypisania nowego, pustego słownika do zmiennej (którą nazwiemy `more_flights`).

```
more_flights = {}
```

1. Zacznij od nowego, pustego słownika.

Określ sposób, w jaki istniejące dane (znajdujące się w słowniku `flights`) mają być iterowane za pomocą pętli `for` (pamiętaj przy tym, aby nie umieszczać na końcu znaku dwukropka).

```
more_flights = {for k, v in flights.items()}
```

2. Iteruj po wszystkich kluczach i wartościach wchodzących w skład istniejących danych.

Zauważ, że NIE ma tu dwukropka.

Aby ukończyć komprehencję słownika, zdefiniuj relację łączącą klucze i wartości nowego słownika. Przedstawiona w górnej części strony pętla `for` generuje klucz nowego słownika, przekształcając klucz starego (zawierający godzinę odlotu w formacie 12-godzinnym) na format 24-godzinny za pomocą funkcji `convert2ampm`. Związana z nią wartość powstaje z kolei w wyniku konwersji wartości pochodzącej ze starego słownika na zapis nazw własnych przy użyciu metody `title` oferowanej przez obiekt łańcucha znakowego. Odpowiadająca tej pętli komprehencja słownika może zrobić to samo, a relację — podobnie jak w przypadku komprehencji list — określa się tu po *lewej* stronie słowa kluczowego `for`. Zwróć tu uwagę na obecność dwukropka oddzielającego nowy klucz od nowej wartości.

```
more_flights = {convert2ampm(k): v.title() for k, v in flights.items()}
```

3. Powiąż skonwertowany klucz z odpowiadającą mu wartością zapisaną jako nazwa własna (i zwróć tu uwagę na zastosowanie dwukropka).

I oto ona: Twoja pierwsza komprehencja słownika. Śmiało, weź ją w obroty, aby przekonać się, czy działa zgodnie z oczekiwaniami.

Rozszerz komprehencje za pomocą filtrów

Wyobraź sobie, że potrzebne są Ci tylko skonwertowane dane dotyczące lotów do miasta *Freeport*.

Gdybyś powrócił do oryginalnej pętli `for`, prawdopodobnie rozszerzyłbyś jej kod tak, aby dodać w nim instrukcję `if`, za pomocą której odfiltrowałbyś dane na podstawie bieżącej wartości przechowywanej w zmiennej `v` (czyli miejsca docelowego). W wyniku tej zmiany otrzymałbyś kod przedstawiony poniżej.

```
just_freeport = {}
for k, v in flights.items():
    if v == 'FREEPORT':
        just_freeport[convert2ampm(k)] = v.title()
```

Dane dotyczące lotów są konwertowane i dodawane do słownika just_freeport tylko, jeśli są związane z miejscem docelowym Freeport.

```
GODZINA,MIEJSCE DOCELOWE
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

Surowe dane

Gdy w wierszu poleceń powłoki Pythona uruchomisz przedstawiony powyżej kod, otrzymasz tylko dwa wiersze danych (reprezentujące dwa loty zaplanowane do *Freeport*, które występują w pliku z surowymi danymi). Nie powinno to być niespodzianką, ponieważ takie zastosowanie instrukcji `if` jest standardowym rozwiązaniem. Okazuje się, że filtrów tego rodzaju można używać również w komprehencjach. Wystarczy, że weźmiesz instrukcję `if` (bez dwukropka) i dołączysz ją na końcu komprehencji. Poniżej znajdziesz komprehencję słownika przedstawioną na końcu poprzedniej strony.

```
more_flights = {convert2ampm(k): v.title() for k, v in flights.items()}
```

Tutaj zaś znajdziesz nową wersję tej komprehencji, w której dodany został odpowiedni filtr.

```
just_freeport2 = {convert2ampm(k): v.title() for k, v in flights.items() if v == 'FREEPORT'}
```

Dane dotyczące lotów są konwertowane i dodawane do słownika just_freeport2 tylko, jeśli są związane z miejscem docelowym Freeport.

Gdy korzystając z wiersza poleceń powłoki Pythona, uruchomisz przedstawiony tu fragment kodu, dane zapisane w nowo utworzonym słowniku `just_freeport2` będą identyczne z tymi, które znajdują się w słowniku `just_freeport`. Dane przechowywane w obydwu tych słownikach są **kopiami** oryginalnych danych umieszczonych w słowniku `flights`.

Oczywiście wiersz kodu generujący słownik `just_freeport2` wygląda dość zniechęcająco. Wielu programistów, dla których Python jest nowym językiem, narzeka, że komprehencje **trudno się czyta**. Przypomnij sobie jednak, że standardowa dla Pythona reguła mówiąca, iż koniec wiersza oznacza koniec instrukcji, nie ma zastosowania tam, gdzie kod pojawia się w nawiasach kwadratowych, dzięki czemu każdą komprehencję możesz zapisać w wielu wierszach, aby łatwiej Ci się ją czytało, tak jak zostało to pokazane poniżej.

```
just_freeport2 = {convert2ampm(k): v.title()
                  for k, v in flights.items()
                  if v == 'FREEPORT'}
```

Będziesz się musiał przyzwyczaić do czytania takich jednowierszowych komprehencji. Warto jednak wiedzieć, że programiści Pythona coraz częściej zapisują dłuższe komprehencje w wielu wierszach (dlatego z tą składnią również będziesz miał nieraz do czynienia).

Przypomnij sobie, co miałeś zrobić

Teraz, gdy wiesz już, co można osiągnąć za pomocą komprehencji, powróćmy do przedstawionych wcześniej w tym rozdziale wymaganych manipulacji na słownikach, aby przekonać się, na jakim jesteśmy etapie. Oto pierwsze z podanych wymagań.

Skonwertuj dane dotyczące czasu z formatu 24-godzinnego na 12-godzinny.

```
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}
```

```
{'09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'
 '05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End' }
```

Skonwertuj zapis danych dotyczących miejsca z WIELKICH LITER na nazwy własne.

Korzystając z danych zapisanych w słowniku `flights`, miałeś okazję zobaczyć, jak pokazana poniżej komprehencja przeprowadza obydwie wymienione powyżej konwersje *w jednym wierszu kodu*, a skopiowane dane są przypisywane do nowego słownika, który nosi tu nazwę `fts`.

```
fts = {convert2ampm(k): v.title() for k, v in flights.items()}
```

Druga manipulacja (utworzenie list godzin odlotów dla każdego miejsca docelowego) wymaga nieco więcej zachodu. Trzeba tu wykonać trochę więcej pracy, ponieważ niezbędne operacje na danych są bardziej skomplikowane.

To, co było wartościami, staje się kluczami.

```
{'09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'
 '05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End' }
```

```
{'Freeport': ['09:35AM', '05:00PM'],
 'Rock Sound': ['11:45AM', '05:55PM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['09:55AM', '07:00PM']}
```

To, co było kluczami, staje się listami wartości.

Zaostrz ołówek

Zanim zabierzemy się do pracy związanej z drugą manipulacją, zatrzymajmy się na chwilę, aby się przekonać, czy materiał dotyczący komprehencji odpowiednio Ci się utrwalił.

Poproszono Cię o przekształcenie w komprehencje trzech pętli `for` znajdujących się na tej stronie. Po zrobieniu tego nie zapomnij przetestować swojego kodu w oknie środowiska IDLE (zanim przejdziesz do kolejnej strony i zapoznasz się z naszymi rozwiązaniami). W gruncie rzeczy najlepiej będzie, jeśli wykonasz te trzy pętle, aby przekonać się, co robią, jeszcze zanim spróbujesz napisać odpowiednie komprehencje. Swoje rozwiązania napisz w pustych miejscach poniżej.

1.
```
data = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
evens = []
for num in data:
    if not num % 2:
        evens.append(num)
```

Operator % to w języku Python operator modulo, którego działanie polega na wykonaniu operacji dzielenia całkowitego pierwszej liczby przez drugą i zwróceniu reszty z tego dzielenia.

2.
```
data = [ 1, 'jeden', 2, 'dwa', 3, 'trzy', 4, 'cztery' ]
words = []
for num in data:
    if isinstance(num, str):
        words.append(num)
```

Funkcja wbudowana isinstance umożliwia sprawdzenie, czy zmienna odwołuje się do obiektu określonego typu.

3.
```
data = list('Cześć, i dzięki za ryby'.split())
title = []
for word in data:
    title.append(word.title())
```

Zaostrz ołówek
Rozwiązanie

Miałeś wziąć do ręki ołówek i zmusić swoje szare komórki do pracy. Twoje zadanie polegało na opracowaniu komprehencji odpowiadających każdej z przedstawionych poniżej pętli `for` oraz przetestowaniu swojego kodu w oknie środowiska IDLE.

1.
```
data = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
evens = []
for num in data:
    if not num % 2:
        evens.append(num)
```

Te cztery wiersze kodu pętli for (odpowiedzialne za wypełnienie listy evens) stały się jednym wierszem komprehencji.

```
evens = [ num for num in data if not num % 2 ]
```

2.
```
data = [ 1, 'jeden', 2, 'dwa', 3, 'trzy', 4, 'cztery' ]
words = []
for num in data:
    if isinstance(num, str):
        words.append(num)
```

I znów — ta czterowierszowa pętla została przekształcona w jednowierszową komprehencję.

```
words = [ num for num in data if isinstance(num, str) ]
```

3.
```
data = list('Cześć, i dzięki za ryby'.split())
title = []
for word in data:
    title.append(word.title())
```

Napisanie tej komprehencji powinno Ci było przyjść najłatwiej (ponieważ nie zawiera ona filtru).

```
title = [ word.title() for word in data ]
```

Poradź sobie z komplikacjami w sposób charakterystyczny dla Pythona

Po wykonaniu naszych ćwiczeń z komprehencjami poeksperymentujmy w wierszu poleceń powłoki Pythona, aby dowiedzieć się, co powinno się stać z danymi zapisanymi w słowniku `fts`, żeby został on przekształcony w wymaganą strukturę danych.

Zanim napiszemy jakikolwiek kod, przyjrzyjmy się pożądanej transformacji. Zwróć uwagą na to, że klucze nowego słownika (widocznego po prawej stronie) tworzą listę unikatowych miejsc docelowych wziętych z wartości słownika `fts` (widocznego po lewej stronie).

To jest słownik fts.

Miejsca docelowe stają się kluczami.

```
{'09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'
 '05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End' }
```

```
{'Freeport': ['09:35AM', '05:00PM'],
 'Rock Sound': ['11:45AM', '05:55PM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['09:55AM', '07:00PM']}
```

Okazuje się, że wygenerowanie tych czterech unikatowych miejsc docelowych jest bardzo proste. Zakładając, że dane pokazane po lewej stronie znajdują się w słowniku o nazwie `fts`, możesz się odwołać do jego wszystkich wartości za pomocą metody `fts.values`, po czym przekazać je funkcji wbudowanej `set`, aby pozbyć się duplikatów. Te unikatowe miejsca docelowe przechowajmy w zmiennej o nazwie `dests`.

Weź wszystkie wartości ze słownika fts, a następnie przekaż je funkcji wbudowanej set. Dzięki temu otrzymasz dane, które są Ci potrzebne.

```
>>> dests = set(fts.values())
>>> print(dests)
{'Freeport', 'West End', 'Rock Sound', 'Treasure Cay'}
```

Oto cztery unikatowe miejsca docelowe, których możesz użyć jako kluczy nowego słownika.

Teraz, gdy dysponujesz już sposobem umożliwiającym uzyskanie unikatowych miejsc docelowych, nadszedł czas na wyciągnięcie godzin odlotów związanych z tymi miejscami. Również te dane znajdują się w słowniku `fts`.

Zanim przejdziesz do kolejnej strony, zastanów się, jak podszedłbyś do zadania polegającego na wydobyciu godzin odlotów dla każdego unikatowego miejsca docelowego.

W gruncie rzeczy nie musisz się martwić wydobywaniem wszystkich godzin odlotów dla *każdego* miejsca docelowego; najpierw wymyśl tylko, jak zrobić to dla miejsca *West End*.

Wyodrębnij godziny odlotów dla jednego miejsca docelowego

Zacznijmy od wyodrębnienia danych dotyczących godzin odlotów dla jednego miejsca docelowego, a mianowicie dla miasta *West End*. Oto dane, które powinieneś wydobyć.

```
{'09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'
 '05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End' }
```

Te klucze powinieneś przetworzyć w listę wartości.

Tak jak wcześniej, przejdź do wiersza poleceń powłoki Pythona i zabierz się do roboty. Mając słownik `fts`, możesz wydobyć godziny odlotów do miasta *West End* za pomocą pokazanego poniżej kodu.

1. Zacznij od nowej, pustej listy.

```
>>> wests = []
>>> for k, v in fts.items():
        if v == 'West End':
            wests.append(k)

>>> print(wests)
['09:55AM', '07:00PM']
```

2. Wydobądź wartości i klucze ze słownika fts.

3. Odfiltruj dane dotyczące miejsca docelowego West End.

4. Dołącz do listy wests godziny odlotów do miasta West End.

Zadziałało! Oto dane, których potrzebujesz.

Widząc ten kod, powinieneś usłyszeć w swojej głowie dźwięk malutkiego dzwonka alarmowego, ponieważ ta pętla `for` jest z pewnością doskonałą kandydatką do stania się komprehencją listy. Zgadza się?
Pętlę tę można przerobić na odpowiadającą jej komprehencję listy w przedstawiony poniżej sposób.

To, co było przed chwilą czterema wierszami kodu, jest teraz jednym wierszem, a to dzięki zastosowaniu komprehencji listy.

```
>>> wests2 = [k for k, v in fts.items() if v == 'West End']

>>> print(wests2)
['09:55AM', '07:00PM']
```

To również zadziałało! Oto dane, których potrzebujesz.

Teraz, gdy wiesz już, jak wydobywać odpowiednie dane dla jednego, konkretnego miejsca docelowego, zróbmy to samo dla wszystkich miejsc.

Wyodrębnij godziny odlotów dla wszystkich miejsc docelowych

W tej chwili masz już kod, za pomocą którego można wydobyć zbiór unikatowych miejsc docelowych.

```
dests = set(fts.values())
```

Unikatowe miejsca docelowe

Masz również komprehencję list umożliwiającą wyodrębnienie godzin odlotów dla danego miejsca docelowego (w naszym przykładzie miejscem tym było miasto *West End*).

Godziny odlotów dla miejsca docelowego West End

```
wests2 = [k for k, v in fts.items() if v == 'West End']
```

Aby wydobyć listę godzin odlotów dla *wszystkich* miejsc docelowych, musisz połączyć te dwa fragmenty kodu (wewnątrz pętli `for`).

W kodzie przedstawionym poniżej pozbyliśmy się zmiennych `dests` i `wests2`, używając zamiast nich odpowiedniego kodu *bezpośrednio* w pętli `for`. Usunęliśmy też zapisane wcześniej na sztywno miejsce *West End*, ponieważ bieżące miejsce docelowe przechowywane jest w zmiennej `dest` (wewnątrz komprehencji list).

Unikatowe miejsca docelowe

```
>>> for dest in set(fts.values()):
        print(dest, '->', [k for k, v in fts.items() if v == dest])

Freeport -> ['09:35AM', '05:00PM']
West End -> ['09:55AM', '07:00PM']
Rock Sound -> ['11:45AM', '05:55PM']
Treasure Cay -> ['10:45AM', '12:00PM']
```

Godziny odlotów dla miejsca docelowego określanego przez bieżącą wartość zmiennej dest

To, że napisaliśmy właśnie pętlę `for`, która wydaje się wpisywać w nasz wzorzec komprehencji, powoduje, że malutki dzwoneczek alarmowy w naszej głowie znów zaczyna dzwonić. Postarajmy się na razie znieść ten dźwięk, ponieważ kod, który właśnie opracowaliśmy w wierszu poleceń powłoki Pythona, *wyświetla* potrzebne nam dane. Jednak tym, czego naprawdę potrzebujemy, jest *zapisanie* tych danych w nowym słowniku. Utwórzmy zatem nowy słownik (o nazwie `when`), który posłuży nam do przechowywania wydobytych niedawno danych. Wróć do wiersza poleceń powłoki Pythona i zmień pętlę `for` w taki sposób, aby została w niej wykorzystana zmienna `when`.

1. Zacznij od nowego, pustego słownika.

2. Wydobądź zbiór unikatowych miejsc docelowych.

```
>>> when = {}
>>> for dest in set(fts.values()):
        when[dest] = [k for k, v in fts.items() if v == dest]

>>> pprint.pprint(when)
{'Freeport': ['09:35AM', '05:00PM'],
 'Rock Sound': ['11:45AM', '05:55PM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['09:55AM', '07:00PM']}
```

3. Zapisz godziny odlotów w słowniku when.

Tutaj mamy to, co jest nam potrzebne: dane znajdują się w słowniku when.

Jeśli jesteś trochę podobny do nas, malutki dzwoneczek w Twojej głowie (ten, który próbowałeś zignorować) prawdopodobnie dzwoni już bardzo głośno i doprowadza Cię do szału, gdy patrzysz na ten kod.

To uczucie, które masz...

...gdy pojedynczy wiersz kodu zaczyna wyglądać jak *czarna magia*.

Wyłącz dzwoneczek w swojej głowie, a następnie jeszcze raz przyjrzyj się kodowi, który tworzy najnowszą wersję naszej pętli `for`.

```
when = {}
for dest in set(fts.values()):
    when[dest] = [k for k, v in fts.items() if v == dest]
```

Kod ten pasuje do naszego wzorca, co czyni go potencjalnym celem przekształcenia w komprehencję. Poniżej znajdziesz ekwiwalent powyższej pętli `for` zapisany jako komprehencja słownika, która wydobywa *kopię* potrzebnych Ci danych i umieszcza ją w nowym słowniku o nazwie `when2`.

```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

Wygląda to na *czarną magię*, nie sądzisz?

To najbardziej skomplikowana komprehencja, z jaką miałeś do czynienia do tej pory. Jej złożoność wynika przede wszystkim z tego, że *zewnętrzna* komprehencja słownika zawiera *wewnętrzną* komprehencję listy. Jednocześnie ta komprehencja słownika pokazuje jedną z charakterystycznych cech odróżniających tego rodzaju konstrukcje od odpowiadających im pętli `for`, a mianowicie to, że komprehencje można umieszczać niemal w dowolnym miejscu w kodzie. Nie jest tak w przypadku pętli `for`, które mogą występować w kodzie wyłącznie jako instrukcje (a więc nie jako części składowe wyrażeń).

```
when = {}
for dest in set(fts.values()):
    when[dest] = [k for k, v in fts.items() if v == dest]
```

Te dwa fragmenty kodu wykonują dokładnie TĘ SAMĄ OPERACJĘ.

```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

Ostrzegamy: komprehencję słownika zawierającą osadzoną komprehencję listy trudno się czyta *za pierwszym razem, gdy ją widzisz*.

Jednak wraz z tym, jak będziesz miał do czynienia z kolejnymi komprehencjami, coraz łatwiej będzie Ci przychodzić ich czytanie i zrozumienie, a — jak stwierdziliśmy to już we wcześniejszej części tego rozdziału — programiści Pythona wykorzystują tego rodzaju konstrukcje *bardzo często*. To, czy będziesz używał komprehencji, zależy wyłącznie od Ciebie. Jeśli czujesz się lepiej z kodem pętli `for`, korzystaj z niego. Jeśli podoba Ci się składnia komprehencji, używaj jej. Nie czuj się jednak do tego *zmuszony*.

Jazda próbna

Zanim przejdziemy dalej, przenieśmy cały kod odpowiedzialny za komprehencję do naszego pliku *do_convert.py*. Możemy następnie uruchomić kod znajdujący się w tym pliku (korzystając ze środowiska IDLE), aby zobaczyć, czy wymagane przez *Brzęczyki z Bahamów* konwersje i transformacje działają zgodnie z oczekiwaniami. Sprawdź, czy Twój kod jest taki sam jak nasz, po czym wykonaj go, aby potwierdzić, że wszystko działa zgodnie ze specyfikacją.

do_convert.py - C:/Users/lsuma/Desktop/Python/R12/do_convert.py (3.6.0)

File Edit Format Run Options Window Help

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

fts = {convert2ampm(k): v.title() for k, v in flights.items()}

pprint.pprint(fts)
print()

when = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}

pprint.pprint(when)
print()
```

Ln: 27 Col: 0

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
========== RESTART: C:/Users/lsuma/Desktop/Python/R12/do_convert.py ==========
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}

{'05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End',
 '09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'}

{'Freeport': ['09:35AM', '05:00PM'],
 'Rock Sound': ['11:45AM', '05:55PM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['09:55AM', '07:00PM']}

>>>
```

Ln: 28 Col: 4

1. Oryginalne, surowe dane w takiej formie, w jakiej zostały odczytane z pliku CSV. To zawartość słownika flights.

2. Surowe dane skopiowane i przekształcone na format 12-godzinny oraz nazw własnych. To zawartość słownika fts.

3. Lista godzin odlotów dla poszczególnych miejsc docelowych (dane wydobyte ze słownika fts). To zawartość słownika when.

Teraz latamy!

Nie istnieją głupie pytania

P: A więc... zapytam wprost: czy komprehencja jest po prostu syntaktycznym skrótem standardowej konstrukcji pętli?

O: Tak, szczególnie pętli `for`. Standardowa pętla `for` i odpowiadającą jej komprehencja robią to samo. Główna różnica polega na tym, że komprehencja jest wykonywana zauważalnie szybciej.

P: Po czym poznać, że należy zastosować komprehencję list?

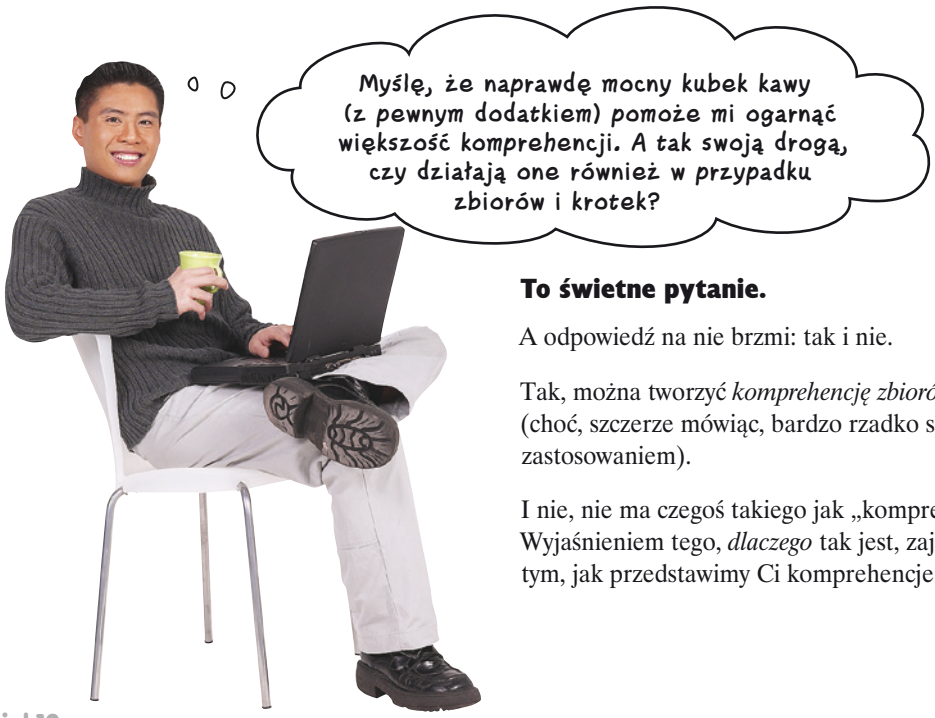
O: Nie ma tu żadnych sztywnych i jednoznacznych reguł. Zazwyczaj, jeśli generujesz nową listę z istniejącej, przyjrzyj się kodowi swojej pętli. Zadaj sobie pytanie, czy jest ona dobrą kandydatką do konwersji na odpowiadającą jej komprehencję. Jeśli nowa lista ma charakter „tymczasowy" (to znaczy, że jest używana tylko raz, a potem porzucana), zadaj sobie pytanie, czy *osadzona* komprehencja listy byłaby lepszym rozwiązaniem w przypadku zadania, z którym masz do czynienia. Za ogólną zasadę powinieneś przyjąć unikanie wprowadzania tymczasowych zmiennych do swojego kodu, zwłaszcza gdy są one używane tylko raz. Zapytaj sam siebie, czy zamiast pętli można w tym miejscu zastosować komprehencję.

P: Czy mogę obyć się zupełnie bez komprehencji?

O: Tak, możesz. Mają one jednak dość duże zastosowanie w powiększającej się społeczności Pythona, jeśli zatem nie zamierzasz konsekwentnie unikać zaglądania w kod innych programistów, sugerujemy Ci poświęcić trochę czasu na zaznajomienie się z mechanizmem komprehencji oferowanym przez język Python. Gdy przyzwyczaisz się do ich widoku, zaczniesz się zastanawiać, jak byłeś w stanie radzić sobie bez nich do tej pory. Czy wspominaliśmy już, że są one wykonywane *szybko*?

P: Tak, rozumiem, ale czy szybkość ma obecnie aż tak wielkie znaczenie? Mam bardzo szybki laptop i wykonuje on pętle `for` wystarczająco prędko.

O: To ciekawe spostrzeżenie. Prawdą jest, że mamy teraz do dyspozycji komputery, które są znacznie wydajniejsze niż cokolwiek, z czym mieliśmy do czynienia w przeszłości. Prawdą jest także to, że poświęcamy obecnie znacznie mniej czasu na próby wyeliminowania ze swojego kodu jak największej liczby zbędnych cykli procesora (ponieważ — powiedzmy sobie szczerze — nie musimy już tego robić). Mając jednak do dyspozycji mechanizm oferujący znaczne zwiększenie wydajności, dlaczego mielibyśmy z niego nie skorzystać? To znaczny zysk wydajności kosztem bardzo niewielkiego wysiłku.



To świetne pytanie.

A odpowiedź na nie brzmi: tak i nie.

Tak, można tworzyć *komprehencję zbiorów* i jej używać (choć, szczerze mówiąc, bardzo rzadko spotkasz się z jej zastosowaniem).

I nie, nie ma czegoś takiego jak „komprehencje krotki". Wyjaśnieniem tego, *dlaczego* tak jest, zajmiemy się po tym, jak przedstawimy Ci komprehencje zbiorów w akcji.

Komprehencja zbioru w akcji

Komprehencje zbiorów (określane też skrótem *setcomp*) umożliwiają tworzenie nowego zbioru za pomocą jednego wiersza kodu, w którym wykorzystuje się konstrukcję bardzo podobną do składni komprehencji listy.

Tym, co odróżnia komprehencje zbiorów od komprehencji list, są otaczające pierwsze nawiasy klamrowe (w przypadku komprehencji list są to oczywiście nawiasy kwadratowe). Może to być nieco mylące, ponieważ komprehencje słowników również otaczają nawiasy klamrowe. (Ciekawe, dlaczego twórcy rdzenia języka Python zdecydowali się na to rozwiązanie).

Literalne zbiory są umieszczane w nawiasach klamrowych, podobnie jak literalne słowniki. Aby stwierdzić, czy dana struktura jest słownikiem, poszukaj znaku dwukropka używanego w słownikach w roli separatora, ponieważ w zbiorach obecność tego znaku nie ma sensu i nie powinien w nich występować. Z tego samego sposobu możesz skorzystać w przypadku konieczności sprawdzenia, czy umieszczona w nawiasach klamrowych komprehencja jest komprehencją słownika czy zbioru; po prostu poszukaj w niej dwukropka. Jeśli go tam znajdziesz, masz przed oczami komprehencję słownika. Jeśli nie, patrzysz na komprehencję zbioru.

Poniżej znajdziesz krótki przykład komprehencji zbioru (odwołujący się do przykładu przedstawionego we wcześniejszej części tego rozdziału). Gdy podamy zbiór liter (zapisany w zmiennej `vowels`) oraz łańcuch znakowy (przechowywany w zmiennej `message`), pętla `for` i odpowiadająca jej komprehencja zbioru generują ten sam wynik, którym jest zbiór samogłosek znalezionych w łańcuchu `message`.

```
vowels = {'a', 'e', 'i', 'o', 'u'}
message = "Nie zapomnij spakować swojego ręcznika."

found = set()
for v in vowels:
    if v in message:
        found.add(v)
```

Komprehencja zbioru naśladuje ten sam wzorzec, z którym jest zgodna z komprehencją listy.

```
found2 = { v for v in vowels if v in message }
```

Zauważ tu zastosowanie nawiasów klamrowych, ponieważ komprehencja ta generuje zbiór, gdy jest wykonywana przez interpreter.

Poświęć kilka chwil na eksperymenty z kodem przedstawionym na tej stronie, które przeprowadzisz w wierszu poleceń powłoki Pythona. Opanowanie komprehencji zbiorów nie powinno stanowić dla Ciebie problemu, ponieważ wiesz już, co można osiągnąć za pomocą komprehencji list i komprehencji słowników. Tak naprawdę nie można o nich powiedzieć wiele więcej, niż zostało zaprezentowane na tej stronie.

Jak dostrzec komprehencję

W miarę zapoznawania się z wyglądem kodu komprehencji coraz łatwiej będzie Ci go dostrzec i zrozumieć. Oto ogólna zasada ułatwiająca rozpoznawanie komprehencji list:

Gdy znajdziesz fragment kodu umieszczony pomiędzy znakami [oraz], wówczas patrzysz na komprehencję listy.

Regułę tę można uogólnić w następujący sposób:

Gdy znajdziesz fragment kodu umieszczony w nawiasach (klamrowych lub kwadratowych), wówczas prawdopodobnie patrzysz na komprehencję.

Dlaczego używamy tu słowa „prawdopodobnie”?

Oprócz nawiasów kwadratowych ([]) komprehencje mogą też otaczać nawiasy klamrowe ({}). Jeśli kod jest umieszczony pomiędzy znakami [oraz], masz do czynienia z komprehencją **listy**. Jeśli znajduje się on pomiędzy znakami { oraz }, patrzysz na komprehencję **zbioru** lub **słownika**. Tę ostatnią łatwo rozpoznać dzięki zastosowaniu w niej znaku dwukropka w roli separatora.

Kod może też jednak występować pomiędzy znakami (i), co jest *szczególnym przypadkiem*, choć nie stanowi bynajmniej *komprehencji krotki*, jak mógłbyś początkowo pomyśleć. Pomysł ten zostałby Ci wybaczony, ale nie jest dobry, ponieważ nie istnieje coś takiego jak „komprehencja krotki”, mimo że kod może być umieszczony w nawiasie okrągłym. Po całej „uciesze”, której dotąd doświadczyłeś z komprehencjami w tym rozdziale, możesz się zastanawiać: *czy to wszystko może być jeszcze dziwniejsze?*

Zakończmy ten rozdział (i całą tę książkę) badaniem tego, co się dzieje z kodem, który pojawia się pomiędzy znakami (oraz). Nie tworzy on „komprehencji krotki”, ale na pewno jest dopuszczalną konstrukcją. Czym zatem jest?

Co z „komprehencjami krotek"?

Cztery wbudowane struktury danych języka Python (krotki, listy, zbiory i słowniki) znajdują bardzo wiele różnych zastosowań. Niemal wszystkie da się też utworzyć za pomocą komprehencji, a wyjątkiem jest tu krotka.

Dlaczego tak jest?

Okazuje się, że idea „komprehencji krotki" tak naprawdę nie ma sensu. Przypomnij sobie, że krotki są *niemodyfikowalne*: gdy zostaną już utworzone, nie mogą być zmieniane. Oznacza to również, że nie da się generować wartości krotki w kodzie, co widać na przykładzie sesji środowiska IDLE pokazanej poniżej.

```
*Python 3.6.0 Shell*
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> names = ()
>>>
>>> for n in ('Jan', 'Paweł', 'Jerzy', 'Ryszard'):
        names.append(n)

Traceback (most recent call last):
  File "<pyshell#12>", line 2, in <module>
    names.append(n)
AttributeError: 'tuple' object has no attribute 'append'
>>>
>>>
                                                Ln: 54  Col: 4
```

Utwórz nową, pustą krotkę.

Spróbuj dynamicznie dodać do niej jakieś dane.

Nie da się dołączyć czegokolwiek do istniejącej krotki, ponieważ jest ona niemodyfikowalna.

Nie dzieje się tu nic dziwnego ani cudownego, ponieważ właśnie takiego zachowania spodziewamy się po krotkach: gdy już istnieją, *nie mogą* zostać zmienione. Już samo to powinno wystarczyć, aby wykluczyć zastosowanie krotek w jakiegokolwiek rodzaju komprehencjach. Przyjrzyj się jednak również przedstawionym poniżej eksperymentom w wierszu poleceń powłoki Pythona. Druga pętla różni się od pierwszej tylko jednym małym szczegółem: nawiasy kwadratowe otaczające komprehencję listy (widoczne w pierwszej pętli) zostały zastąpione nawiasami okrągłymi (w przypadku drugiej pętli).

```
Python 3.6.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> for i in [x*3 for x in [1,2,3,4,5]]:
        print(i)


3
6
9
12
15
>>>
>>> for i in (x*3 for x in [1,2,3,4,5]):
        print(i)


3
6
9
12
15
>>>
                                                Ln: 74  Col: 4
```

To połączenie pętli for i komprehencji listy wyświetla każdą wartość należącą do listy pomnożoną przez trzy. Wiesz, że masz tu do czynienia z komprehencją listy, ponieważ kod ten znajduje się w nawiasie kwadratowym.

Co jest grane? Obydwie pętle generują te same wyniki.

Spójrz jednak na to. Nawias okrągły powoduje, że wygląda to na „komprehencję krotki", wiesz jednak, że taka konstrukcja jest niemożliwa. Mimo to przedstawiona tu pętla generuje dane, których się spodziewasz. Dziwne, nie?

Nawiasy okrągłe wokół kodu = generator

Jeśli natrafiasz na coś, co wygląda jak komprehencja listy, lecz jest ujęte w nawiasy okrągłe, masz do czynienia z **generatorem**.

To wygląda jak komprehencja listy, ale nią nie jest; jest za to generatorem.

```
for i in (x*3 for x in [1,2,3,4,5]):
    print(i)
```

Generator może być używany wszędzie tam, gdzie używana jest komprehencja listy, generuje również te same wyniki co ona.

Jak widać to na końcu poprzedniej strony, jeśli nawiasy kwadratowe obejmujące komprehencję listy zastąpisz nawiasami okrągłymi, otrzymasz te same wyniki; oznacza to, że generator i komprehencja listy generują te same dane.

Nie są jednak wykonywane w ten sam sposób.

Jeśli zastanawiasz się teraz nad ostatnim zdaniem, weź pod uwagę to, że gdy wykonywana jest komprehencja listy, **wszystkie** jej dane generowane są przed jakimkolwiek innym przetwarzaniem. Jeśli odniesiemy to do przykładu przedstawionego na początku tej strony, będzie to znaczyło, że pętla `for` nie rozpocznie przetwarzania *żadnych* danych wytworzonych przez komprehencję listy, dopóki ta ostatnia nie zostanie ukończona. Oznacza to, że komprehencja listy, która generuje swoje dane przez długi czas, powoduje opóźnienie wykonania wszelkiego innego kodu aż do momentu, *gdy zostanie zakończona*.

W przypadku tej niewielkiej listy elementów danych (która została pokazana powyżej) nie stanowi to wielkiego problemu.

Wyobraź sobie jednak, że Twoja komprehencja listy ma tworzyć listę generującą 10 milionów elementów danych. Masz w takiej sytuacji do czynienia z dwoma problemami: po pierwsze, musisz czekać aż komprehencja listy przetworzy 10 milionów elementów danych, *zanim będziesz mógł zrobić coś innego*, a po drugie, musisz martwić się o to, czy komputer, na którym uruchamiasz swój kod, ma wystarczająco dużo pamięci operacyjnej, aby zmieścić w niej wszystkie dane niezbędne do wykonania komprehencji (**10 milionów** oddzielnych kawałków danych). Gdy Twoja komprehencja listy wyczerpie dostępną dla niej pamięć operacyjną, interpreter przerwie swoje działanie (a Twój program pójdzie z dymem).

Komprehencje list i generatory wytwarzają te same wyniki, ale działają w zupełnie inny sposób.

Generatory wytwarzają elementy danych po jednym naraz...

Gdy zastąpisz nawiasy kwadratowe swojej komprehencji listy nawiasami okrągłymi, komprehencja ta stanie się **generatorem**, a Twój kod zacznie zachowywać się inaczej.

W przeciwieństwie do komprehencji list, które muszą się zakończyć, zanim będzie mógł zostać wykonany jakikolwiek inny kod, generator zwraca dane od razu po ich wytworzeniu. Oznacza to, że gdy będziesz generował 10 milionów elementów danych, interpreter będzie potrzebował tylko tyle pamięci, aby umieścić **jeden** element (naraz), a jakikolwiek kod oczekujący na konsumpcję elementów danych tworzonych przez generator będzie mógł być wykonany natychmiast. A więc *nie ma tu żadnego czekania*.

Różnicy, jaką może zrobić zastosowanie generatora, nie da się wyjaśnić w lepszy sposób niż za pomocą przykładu, dlatego wykonajmy pewne proste zadanie dwukrotnie: najpierw za pomocą komprehencji listy, a potem przy użyciu generatora.

Używanie komprehencji listy do przetwarzania URL-i

W celu zaprezentowania różnicy, jaką może zrobić zastosowanie generatora, przeprowadźmy pewną operację za pomocą komprehencji listy (zanim zrobimy to samo, korzystając z generatora).

Pobierz bibliotekę requests z serwisu PyPI, korzystając z polecenia pip.

Jak mieliśmy w zwyczaju w całej tej książce, poeksperymentujmy trochę w wierszu poleceń powłoki Pythona z kodem, w którym tym razem wykorzystamy bibliotekę `requests` (umożliwiającą interakcję z siecią z poziomu programu). Poniżej znajdziesz krótką sesję, w której importujemy tę bibliotekę, definiujemy trójelementową krotkę (o nazwie `urls`), a następnie za pomocą kombinacji pętli `for` oraz komprehencji listy żądamy dostępu do docelowej strony wskazywanej przez każdy URL przed przetworzeniem zwróconej odpowiedzi sieciowej.

Aby zrozumieć, co się tu dzieje, powinieneś wykonywać na swoim komputerze kroki, które zaprezentujemy poniżej.

Zdefiniuj krotkę zawierającą ciągi URL. Nie krępuj się, jeśli chcesz użyć w tym miejscu innych URL-i. Pamiętaj tylko, żeby podać co najmniej trzy.

Pętla for zawiera komprehencję listy, która pobiera docelową stronę dla każdego URL-a zapisanego w krotce urls.

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> import requests
>>>
>>> urls = ('http://helion.pl', 'http://onepress.pl', 'http://septem.pl')
>>>
>>> for resp in [requests.get(url) for url in urls]:
        print(len(resp.content), '->', resp.status_code, '->', resp.url)


243434 -> 200 -> http://helion.pl/
226913 -> 200 -> http://onepress.pl/
55083 -> 200 -> http://septem.pl/

>>> |
```

Ln: 256 Col: 4

Nie dzieje się tu nic dziwnego ani cudownego. Otrzymane dane wyjściowe dokładnie odpowiadają oczekiwaniom.

Dla każdej otrzymanej odpowiedzi wyświetl (wyrażoną w bajtach) wielkość zwróconej strony docelowej, kod stanu HTTP oraz użyty URL.

Jeśli zrobisz na swoim komputerze to samo, co my tutaj, doświadczysz zauważalnego opóźnienia pomiędzy wprowadzeniem kodu pętli `for` i wyświetleniem wyników. Gdy na ekranie pojawią się wyniki, zostaną one wyświetlone za jednym zamachem (czyli wszystkie naraz). Dzieje się tak, ponieważ komprehencja listy wykonuje operacje na każdym URL-u przechowywanym w krotce `urls`, zanim udostępni pętli `for` jakiekolwiek wyniki. Jaki jest tego efekt? Taki, że musisz czekać na wyświetlenie danych wyjściowych.

Zauważ, że nie ma *nic* niewłaściwego w tym kodzie: robi to, czego od niego oczekujemy, a generowane dane wyjściowe są prawidłowe. Przeróbmy jednak tę komprehencję listy na generator, aby przekonać się, jaką zrobi to różnicę. Jak wspomnieliśmy już powyżej, powinieneś naśladować nasze działania na swoim komputerze — wykonuj zatem operacje przedstawione na następnej stronie (abyś mógł się przekonać, co się stanie).

Używanie generatora do przetwarzania URL-i

Poniżej znajdziesz przykład zaprezentowany na poprzedniej stronie przerobiony w taki sposób, aby wykorzystywał generator. Wykonanie odpowiedniej zmiany jest bardzo proste — wystarczy tu tylko zastąpić nawiasy kwadratowe komprehencji listy nawiasami okrągłymi.

```
*Python 3.6.0 Shell*
File Edit Shell Debug Options Window Help
>>>
>>>
>>>
>>>
>>> for resp in (requests.get(url) for url in urls):
        print(len(resp.content), '->', resp.status_code, '->', resp.url)
Ln: 264 Col: 1
```

Istotna zmiana: nawiasy kwadratowe zostały zastąpione nawiasami okrągłymi.

Krótko po wprowadzeniu powyższej pętli `for` na ekranie pojawia się pierwszy wynik.

```
*Python 3.6.0 Shell*
File Edit Shell Debug Options Window Help
>>>
>>>
>>>
>>> for resp in (requests.get(url) for url in urls):
        print(len(resp.content), '->', resp.status_code, '->', resp.url)

243434 -> 200 -> http://helion.pl/
```

Odpowiedź dla pierwszego URL-a

Następnie, chwilę później, na ekranie pojawia się drugi wiersz wyników.

```
*Python 3.6.0 Shell*
File Edit Shell Debug Options Window Help
>>>
>>>
>>> for resp in (requests.get(url) for url in urls):
        print(len(resp.content), '->', resp.status_code, '->', resp.url)

243434 -> 200 -> http://helion.pl/
226913 -> 200 -> http://onepress.pl/
```

Odpowiedź dla drugiego URL-a

Następnie — na koniec — po kolejnej chwili na ekranie pojawia się ostatni wiersz wyników (a pętla `for` kończy swoje działanie).

```
*Python 3.6.0 Shell*
File Edit Shell Debug Options Window Help
>>>
>>> for resp in (requests.get(url) for url in urls):
        print(len(resp.content), '->', resp.status_code, '->', resp.url)

243434 -> 200 -> http://helion.pl/
226913 -> 200 -> http://onepress.pl/
55083 -> 200 -> http://septem.pl/
>>>
Ln: 269 Col: 21
```

Odpowiedź dla trzeciego (i ostatniego) URL-a

Używanie generatora: co się właśnie stało?

Jeśli porównasz wyniki wytworzone przez komprehencję listy do tych otrzymanych w wyniku działania generatora, okaże się, że są one *identyczne*. Jednak zachowanie tych dwóch konstrukcji takie nie jest.

Komprehencja listy **czeka** na wytworzenie wszystkich swoich danych przed przekazaniem czegokolwiek oczekującej pętli `for`, podczas gdy generator **uwalnia** dane od razu, gdy tylko staną się dostępne. Oznacza to, że pętla `for` wykorzystująca generator jest znacznie bardziej responsywna niż analogiczna pętla, w której użyto komprehencji listy (sprawiającej, że czekasz).

Jeśli wydaje Ci się, że nie ma to wielkiego znaczenia, wyobraź sobie, że krotka z URL-ami zawiera sto, tysiąc lub milion elementów. Dalej, wyobraź sobie, że kod przetwarzający odpowiedź przekazuje zmodyfikowane dane innemu procesowi (na przykład oczekującej bazie danych). Wraz ze wzrostem liczby URL-i zachowanie kodu wykorzystującego komprehencję listy staje się coraz gorsze w porównaniu z tym, jak działa kod używający generatora.

Nie. Tego byśmy nie powiedzieli.

Nie zrozum nas źle: to, że istnieją generatory, jest *wspaniałe*, ale nie oznacza, że powinieneś zastąpić swoje wszystkie komprehencje listy odpowiadającymi im generatorami. Podobnie jak z innymi kwestiami dotyczącymi programowania, również tutaj wybór właściwego sposobu mocno zależy od tego, co próbujesz zrobić.

Jeśli możesz sobie pozwolić na oczekiwanie, wówczas komprehencje listy są w porządku; w innym razie rozważ zastosowanie generatora.

Interesującym zastosowaniem generatora jest osadzenie go wewnątrz funkcji. Przyjrzyjmy się sposobowi umożliwiającemu hermetyzację w funkcji utworzonego przez nas właśnie generatora.

Zdefiniuj, co powinna robić Twoja funkcja

Wyobraźmy sobie, że chcesz przekształcić swój generator żądań w funkcję. Postanowiłeś opakować go w niewielki moduł, który piszesz, chciałbyś też, aby inni programiści mogli z niego korzystać bez konieczności rozumienia działania generatorów czy posiadania jakiejkolwiek wiedzy na ich temat.

Poniżej został jeszcze raz pokazany kod Twojego generatora.

```
import requests

urls = ('http://helion.pl', 'http://onepress.pl', 'http://septem.pl')

for resp in (requests.get(url) for url in urls):
    print(len(resp.content), '->', resp.status_code, '->', resp.url)
```

Zaimportuj wszystkie niezbędne biblioteki.

Zdefiniuj krotkę z URL-ami.

Przetwórz wygenerowane dane.

Generator (pamiętaj, że wygląda jak komprehencja listy, ale jest umieszczony w nawiasach okrągłych).

Utwórzmy funkcję, która hermetyzuje ten kod. Funkcja ta, nosząca nazwę `gen_from_urls`, przyjmuje pojedynczy argument (krotkę zawierającą URL-e) i zwraca krotkę wyników dla każdego otrzymanego URL-a. Zwrócona krotka zawiera trzy wartości: wielkość treści dostępnej pod URL-em, kod stanu HTTP oraz URL, spod którego nadeszła odpowiedź.

Po opracowaniu funkcji `gen_from_urls` będziesz chciał, aby inni programiści mogli ją uruchamiać jako część pętli `for`, tak jak zostało to pokazane poniżej.

```
from url_utils import gen_from_urls

urls = ('http://helion.pl', 'http://onepress.pl', 'http://septem.pl')

for resp_len, status, url in gen_from_urls(urls):
    print(resp_len, status, url)
```

Zaimportuj funkcję ze swojego modułu.

Zdefiniuj krotkę z URL-ami.

Wywołaj funkcję w każdej iteracji pętli for.

Przetwórz dane.

Choć ten nowy kod nie różni się zbytnio od kodu przedstawionego na początku tej strony, zauważ, że programiści korzystający z funkcji `gen_from_urls` nie mają zielonego pojęcia (ani nie muszą wiedzieć), że korzystasz z biblioteki `requests`, aby komunikować się z siecią. Nie muszą też wiedzieć, że korzystasz z generatora. Wszystkie szczegóły implementacji i decyzje, jakie podjąłeś, są ukryte za tym łatwym do zrozumienia wywołaniem funkcji.

Zobaczmy, z czym wiąże się pisanie kodu funkcji `gen_from_urls` w taki sposób, aby generowała potrzebne Ci dane.

Ulegnij potędze funkcji generatorowych

Teraz, gdy wiesz już, co powinna robić funkcja `gen_from_urls`, zajmijmy się jej pisaniem. Zacznij od utworzenia nowego pliku o nazwie *url_utils.py*. Otwórz go do edycji, a następnie jako pierwszy wiersz kodu dodaj instrukcję `import requests`.

Wiersz `def` naszej funkcji jest bardzo prosty, ponieważ przyjmuje ona jedną krotkę jako argument i zwraca krotkę jako swój wynik (zauważ, że dołączyliśmy adnotacje typu, aby było to jasne dla użytkowników tej funkcji generatorowej). Śmiało, dodaj wiersz `def` funkcji do pliku.

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
```

Po zaimportowaniu modułu requests zdefiniuj swoją nową funkcję.

Zestawem funkcji jest generator przedstawiony na poprzedniej stronie, a wiersz zawierający pętlę `for` to kod skopiowany wprost z naszego ostatniego przykładu.

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
```

Dodaj wiersz pętli for zawierający generator.

Następny wiersz kodu powinien „zwracać” wynik żądania `GET` przetworzonego przez funkcję `requests.get`. Mimo że może Cię kusić dodanie pokazanego poniżej wiersza kodu jako zestawu pętli `for`, **prosimy, nie rób tego**.

```
return len(resp.content), resp.status_code, resp.url
```

Gdy funkcja wykonuje instrukcję `return`, działanie funkcji *się kończy*. Nie chcesz, żeby stało się to tutaj, ponieważ funkcja `gen_from_urls` jest wywoływana jako część pętli `for`, która spodziewa się *innej* krotki *za każdym razem, gdy funkcja jest wywoływana*.

Skoro jednak nie możesz wykonać instrukcji `return`, to co masz zrobić?

Zamiast niej użyj instrukcji `yield`. Słowo kluczowe `yield` zostało dodane do języka Python, aby zapewniać możliwość tworzenia **funkcji generatorowych**, i możesz używać go wszędzie tam, gdzie wykorzystywana jest instrukcja `return`. Gdy z niego korzystasz. Twoja funkcja przekształca się w funkcję generatorową, która może być „wywoływana” z dowolnego iteratora, a więc — w tym przypadku — z wnętrza pętli `for`.

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Użyj instrukcji yield, aby zwracać każdy wiersz wyników z odpowiedzi GET do oczekującej pętli for. Pamiętaj, NIE korzystaj tu z instrukcji return.

Przyjrzyjmy się bliżej temu, co się tu dzieje.

Śledzenie Twojej funkcji generatorowej (1. z 2)

Aby zrozumieć, co się dzieje podczas działania Twojej funkcji generatorowej, prześledźmy wykonanie następującego kodu:

Zaimportuj swoją funkcję generatorową.

Zdefiniuj krotkę z URL-ami.

```
from url_utils import gen_from_urls

urls = ('http://helion.pl', 'http://onepress.pl', 'http://septem.pl')

for resp_len, status, url in gen_from_urls(urls):
    print(resp_len, status, url)
```

Użyj swojej funkcji generatorowej jako części pętli for.

Dwa pierwsze wiersze tego kodu są bardzo proste: funkcja zostaje zaimportowana, a krotka z URL-ami — zdefiniowana.

Zabawa zaczyna się w następnym wierszu, gdy funkcja generatorowa `gen_from_urls` jest wywoływana. Tę pętlę `for` będziemy określać mianem „kodu wywołującego”.

```
for resp_len, status, url in gen_from_urls(urls):
```

Pętla for należąca do kodu wywołującego komunikuje się z pętlą for należącą do funkcji generatorowej.

Interpreter przeskakuje do funkcji `gen_from_urls` i zaczyna wykonywać jej kod. Krotka zawierająca URL-e jest kopiowana do jedynego argumentu tej funkcji, po czym wykonywana jest pętla `for` należąca do funkcji generatorowej.

```
def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Ta pętla `for` zawiera generator, który pobiera pierwszy URL z krotki `urls` i wysyła żądanie `GET` do zidentyfikowanego serwera. Gdy odpowiedź HTTP zostaje zwrócona przez serwer, wykonywana jest instrukcja `yield`.

Dotarliśmy do miejsca, w którym sprawy robią się ciekawe (lub dziwne, zależnie od Twojego punktu widzenia).

Zamiast po wykonaniu przechodzić do następnego URL-a znajdującego się w krotce `urls` (a więc kontynuować działanie w następnej iteracji pętli `for` należącej do funkcji `gen_from_urls`), instrukcja `yield` przekazuje swoje trzy elementy danych z powrotem do kodu wywołującego. Zamiast kończyć swoje działanie, funkcja generatorowa `gen_from_urls` teraz *oczekuje*, jakby popadła w pewien rodzaj *letargu*...

Śledzenie Twojej funkcji generatorowej (2. z 2)

Gdy dane (przekazane z powrotem przez instrukcję `yield`) dotrą do wywołującego kodu, zestaw pętli `for` zostaje wykonany. Jako że zestaw ten zawiera pojedyncze wywołanie wbudowanej funkcji `print`, odpowiedni wiersz kodu zostaje wykonany i wyświetla na ekranie wyniki dla pierwszego URL-a.

```
print(resp_len, '->', status, '->', url)
```

```
243434 -> 200 -> http://helion.pl/
```

Następnie iteruje pętla `for` należąca do kodu wywołującego, ponownie wywołując funkcję `gen_from_urls`... czy jakoś tak.

To *prawie* to, co się tu dzieje. Bowiem tak naprawdę funkcja `gen_from_urls` zostaje wybudzona z letargu, po czym kontynuuje działanie. Pętla `for` znajdująca się wewnątrz funkcji `gen_from_urls` iteruje, pobiera kolejny URL z krotki `urls` i nawiązuje kontakt z serwerem związanym z tym URL-em. Gdy odpowiedź HTTP zostaje zwrócona przez serwer, wykonywana jest instrukcja `yield`, która przekazuje trzy elementy danych z powrotem do wywołującego kodu (funkcja ta odwołuje się do nich, korzystając z obiektu `resp`).

```
yield len(resp.content), resp.status_code, resp.url
```

Trzy dostarczone elementy danych zostają wzięte z obiektu resp zwróconego przez metodę get należącą do biblioteki requests.

Tak jak wcześniej, zamiast kończyć swoje działanie, funkcja generatorowa `gen_from_urls` po raz kolejny *oczekuje*, jakby wpadła w stan *letargu*...

Gdy dane (przekazane z powrotem przez instrukcję `yield`) dotrą do wywołującego kodu, po raz kolejny zostaje wykonany zestaw pętli `for` zawierający wywołanie funkcji `print`, co powoduje wyświetlenie na ekranie drugiego zbioru wyników.

```
226913 -> 200 -> http://onepress.pl/
55083 -> 200 -> http://septem.pl/
```

Pętla `for` należąca do kodu wywołującego iteruje, po raz kolejny „wywołując” funkcję `gen_from_urls`, co skutkuje ponownym wybudzeniem Twojej funkcji generatorowej. Wykonana zostaje instrukcja `yield`, wyniki są zwracane do wywołującego kodu, a na ekranie pojawiają się kolejne informacje.

```
243434 -> 200 -> http://helion.pl/
226913 -> 200 -> http://onepress.pl/
55083 -> 200 -> http://septem.pl/
```

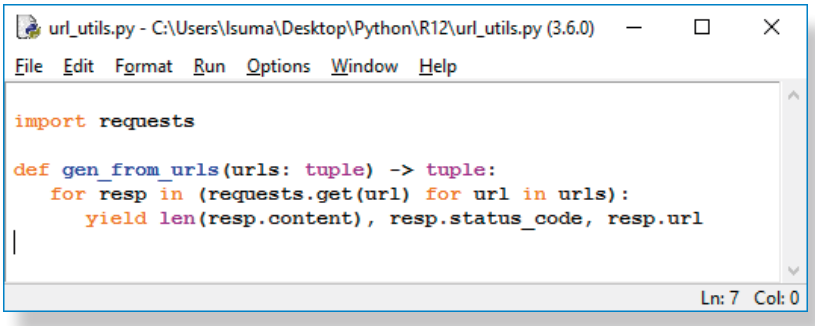
W tym momencie Twoja krotka z URL-ami zostaje wyczerpana, dlatego zarówno funkcja generatorowa, jak i pętla `for` należąca do wywołującego kodu kończą swoją pracę. To tak, jakby dwa kawałki kodu czekały na swoją kolej, aby zostać wykonane, przekazując sobie nawzajem odpowiednie dane w każdym cyklu.

Przyjrzyjmy się działaniu tego rozwiązania w wierszu poleceń powłoki Pythona. Nadszedł czas na ostatnią *Jazdę próbną*.

Jazda próbna

W ramach tej, ostatniej w książce *Jazdy próbnej* weźmy w obroty naszą funkcję generatorową. Tak jak robiłeś to od samego początku, załaduj swój kod w oknie edycji środowiska IDLE, po czym naciśnij przycisk *F5*, aby sprawdzić działanie tej funkcji w wierszu poleceń powłoki Pythona. Staraj się naśladować nasze (przedstawione poniżej) testy.

Oto kod funkcji generatorowej gen_from_urls zapisanej w module url_utils.py.

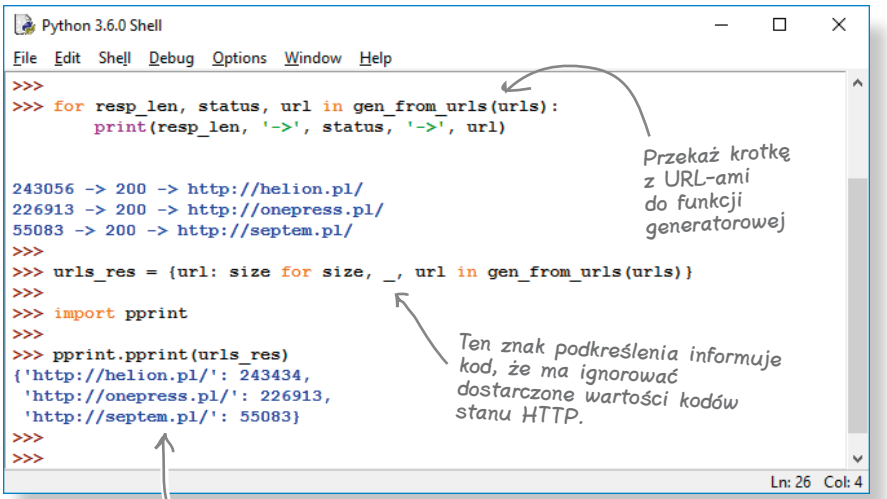


Pierwszy przedstawiony poniżej przykład pokazuje wywołanie funkcji `gen_from_urls` jako część pętli `for`. Jak można się tego spodziewać, efekt działania tego fragmentu kodu jest taki sam jak ten, który udało nam się uzyskać kilka stron wcześniej.

Drugi zamieszczony poniżej przykład pokazuje zastosowanie funkcji `gen_from_urls` jako części komprehencji słownika. Zwróć uwagę, że nowy słownik musi przechowywać jedynie URL-e (jako klucze) oraz wielkości docelowych stron internetowych (jako wartości). W tym przykładzie nie jest potrzebny kod stanu HTTP, dlatego każemy interpreterowi go zignorować, korzystając w tym celu ze standardowej w Pythonie nazwy zmiennej (którą jest pojedynczy znak podkreślenia).

Każdy wiersz wyników pojawia się po krótkiej przerwie, ponieważ dane są generowane przez funkcję.

Ta komprehencja słownika wiąże URL z wielkością jego strony docelowej.



Przekaż krotkę z URL-ami do funkcji generatorowej

Ten znak podkreślenia informuje kod, że ma ignorować dostarczone wartości kodów stanu HTTP.

Ładne wyświetlenie słownika urls_res potwierdza, że funkcja generatorowa może być wykorzystywana wewnątrz komprehencji słownika (jak również wewnątrz pętli for).

Uwagi końcowe

Używanie komprehencji i funkcji generatorowych jest zwykle uważane za zaawansowany temat w świecie Pythona. Jest tak jednak głównie dlatego, że możliwości tych brakuje w innych popularnych językach programowania, co oznacza, że programiści zaczynający korzystać z Pythona czasami mają problem ze zrozumieniem działania tego rodzaju konstrukcji (ponieważ nie mają żadnego punktu odniesienia, który mógłby im to ułatwić).

Niemniej jednak zespół programistów Pythona pracujący w laboratorium *Rusz głową!* po prostu *uwielbia* komprehencje i generatory, a także szczerze wierzy, że dzięki powtarzającemu się kontaktowi z konstrukcjami tego rodzaju definiowanie pętli, w których są one wykorzystywane, z czasem staje się drugą naturą. Wręcz nie wyobrażają sobie życia bez tych możliwości.

Nawet jeśli uznasz składnię komprehencji i generatorów za dziwną, radzimy Ci z nich korzystać. Nawet jeśli zignorujesz to, że są one wydajniejsze niż odpowiadające im pętle `for`, do używania tych konstrukcji Pythona powinien przekonać Cię fakt, że komprehencje i generatory możesz stosować w miejscach, w których nie możesz korzystać z pętli `for`. Z czasem, gdy oswoisz się już z ich składnią, okazje do zastosowania komprehencji i generatorów będą pojawiać się tak naturalnie jak te, które każą Twojemu programistycznemu umysłowi w jednym miejscu użyć funkcji, w innym pętli, w jeszcze innym klasy i tak dalej. Poniżej znajdziesz przegląd tego, czego się dowiedziałeś w tym rozdziale.

CELNE SPOSTRZEŻENIA

- Język Python oferuje szereg opcji związanych z korzystaniem z danych zapisanych w plikach. Oprócz standardowej funkcji wbudowanej `open` możesz tu użyć możliwości oferowanych przez moduł `csv` należący do biblioteki standardowej, które pozwalają Ci wygodnie posługiwać się danymi w formacie CSV.
- **Łańcuchy** metod umożliwiają przetwarzanie danych w pojedynczym wierszu kodu. Łańcuch `string.strip().split()` niezwykle często występuje w kodzie języka Python.
- Bądź ostrożny, ustalając kolejność metod w łańcuchu. Zwróć szczególną uwagę na typ danych zwracanych przez każdą z tych metod (i upewnij się, że zachowana jest zgodność typów).
- Pętla `for` używana do zmiany formatu danych może zostać przerobiona na komprehencję.
- Komprehencje mogą być wykorzystywane do przetwarzania istniejących list, słowników oraz zbiorów, przy czym komprehencje list są najbardziej popularnym rodzajem spotykanym „w naturze". Doświadczeni programiści Pythona mówiąc o tych konstrukcjach, używają często nazw *listcomp*, *dictcomp* oraz *setcomp*.
- **Komprehencja listy** to kod umieszczony w nawiasach kwadratowych, podczas gdy **komprehencja słownika** to kod otoczony nawiasami klamrowymi (i zawierający dwukropki jako separatory). **Komprehencja zbioru** to także kod umieszczony w nawiasach klamrowych (ale pozbawiony separatorów).
- Nie istnieje coś takiego jak „komprehencja krotki", ponieważ krotki są niemodyfikowalne (a zatem nie ma sensu próbować tworzyć ich w sposób dynamiczny).
- Jeśli znajdziesz kod komprehencji umieszczony w nawiasach okrągłych, oznacza to, że masz do czynienia z **generatorem** (który można przerobić na funkcję wykorzystującą instrukcję `yield` do generowania danych zgodnie z potrzebą).

Jako że ten rozdział dobiega końca (jak również, z definicji, główna treść tej książki), mamy do Ciebie ostatnie pytanie. Weź głęboki oddech, a następnie przejdź na następną stronę.

Ostatnie pytanie

W porządku, oto ono, nasze ostatnie pytanie do Ciebie: czy po dotarciu do tego miejsca w książce *w ogóle zwracasz jeszcze uwagę na stosowanie znaczących białych znaków w języku Python?*

Programiści, dla których Python jest nowym językiem, narzekają najczęściej na używanie w nim białych znaków do oznaczania bloków kodu (zamiast wykorzystywania w tym celu na przykład nawiasów klamrowych). Jednak po pewnym czasie Twój mózg zupełnie przestaje zwracać na to uwagę.

To nie przypadek: użycie białych znaków w Pythonie zostało celowo zdefiniowane przez twórcę tego języka w taki, a nie inny sposób.

Zrobiono to z rozmysłem, ponieważ **kod czyta się częściej, niż pisze**. Oznacza to, że kod pasujący wyglądem do spójnego i dobrze znanego wzorca okazuje się łatwiejszy do odczytania. Znaczy to również, że kod Pythona napisany 10 lat temu przez zupełnie obcą Ci osobę jest dla Ciebie czytelny *dzisiaj* dzięki sposobowi użycia białych znaków w tymże języku.

Kod przykładów z rozdziału 12.

To jest zawartość pliku do_convert.py.

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
   return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
   ignore = data.readline()
   flights = {}
   for line in data:
      k, v = line.strip().split(',')
      flights[k] = v

pprint.pprint(flights)
print()

fts = {convert2ampm(k): v.title() for k, v in flights.items()}

pprint.pprint(fts)
print()

when = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}

pprint.pprint(when)
print()
```

To jest zawartość pliku url_utils.py.

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
   for resp in (requests.get(url) for url in urls):
      yield len(resp.content), resp.status_code, resp.url
```

Czas w drogę...

Jesteś w drodze!

Smutno nam, że odchodzisz, lecz nic nie uczyni nas szczęśliwszymi niż to, że weźmiesz ze sobą wszystko, czego nauczyłeś się z tej książki o Pythonie, i *zaczniesz stosować tę wiedzę w praktyce.* Znajdujesz się na początku swojej podróży z Pythonem i zawsze będzie coś jeszcze do poznania. Oczywiście nie skończyłeś jeszcze lektury tej książki. Jest w niej jeszcze pięć (tak, pięć!) dodatków, które powinieneś przeczytać. Obiecujemy, że nie są zbyt długie i zdecydowanie warto się z nimi zapoznać. Jest tu też, oczywiście, indeks — nie zapominajmy o nim!

Mamy nadzieję, że nauka Pythona była dla Ciebie równie dobrą zabawą, jaką dla nas było pisanie tej książki. To był naprawdę wspaniały czas. Baw się dobrze!

Instalowanie Pythona

Najpierw rzeczy najważniejsze: zainstalujmy Pythona na Twoim komputerze.

Pythona możesz swobodnie używać niezależnie od tego, czy korzystasz z systemu operacyjnego *Windows*, *Mac OS X* czy *Linux*. Sposób instalacji na tych platformach różni się jednak nieco w związku z odmiennym działaniem poszczególnych systemów (wiemy, ta wiadomość jest dla Ciebie nie lada szokiem, prawda?), lecz społeczność Pythona ciężko pracuje, aby zapewnić programy instalacyjne współpracujące ze wszystkimi popularnymi środowiskami operacyjnymi. W tym krótkim dodatku przeprowadzimy Cię przez procedurę instalacji Pythona na Twoim komputerze.

Zainstaluj Pythona 3 w systemie Windows

Jeśli nie zainstalowałeś interpretera języka Python na swoim komputerze osobistym pracującym pod kontrolą systemu operacyjnego *Windows* (lub nie zrobił tego ktoś inny), jest bardzo mało prawdopodobne, aby był on preinstalowany. Jednak nawet jeśli jest, zainstalujmy teraz najnowszą i najlepszą wersję Pythona 3 na Twoim komputerze z *Windowsem*.

Jeśli masz już zainstalowaną wersję Pythona 3, zostanie ona zaktualizowana. Jeśli masz zainstalowanego Pythona 2, Python 3 zostanie zainstalowany obok niego (ale nie będzie w żaden sposób wchodził mu w drogę). A jeśli nie masz jeszcze żadnej wersji Pythona, wkrótce się to zmieni!

Pobierz, następnie zainstaluj

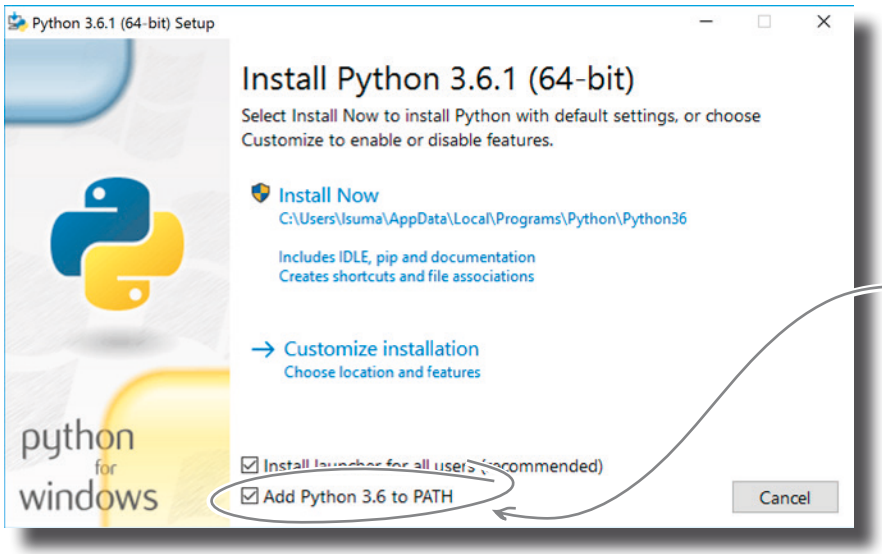
Korzystając ze swojej przeglądarki internetowej, przejdź na stronę *https://www.python.org*, po czym kliknij widoczną na niej kartę *Downloads* (rzeczy do pobrania).

W oknie przeglądarki pojawią się dwa wielkie przyciski umożliwiające wybór najnowszej wersji Pythona 3 lub Pythona 2. *Kliknij przycisk Pythona 3*. Wybierz zapis pliku na dysku, gdy zostaniesz o to poproszony. Po pewnym czasie operacja pobierania zostanie zakończona. Znajdź pobrany plik w swoim folderze *Pobrane* (*Download* lub w innym miejscu, w którym zapisałeś plik), po czym kliknij go dwukrotnie, aby rozpocząć instalację.

W wyniku tego zacznie się proces instalacji standardowy dla systemu *Windows*. W większości kroków kreatora pojawiających się na ekranie możesz po prostu naciskać przycisk *Next* (dalej), wyjątkiem będzie krok pokazany poniżej, w którym powinieneś zatrzymać się na chwilę i wprowadzić drobną zmianę w konfiguracji instalacji. Chodzi tu o zaznaczenie pola wyboru *Add Python 3.6 to PATH* (dodaj Pythona 3.6 do zmiennej środowiskowej PATH), dzięki czemu system *Windows* będzie mógł znaleźć interpreter zawsze, gdy będzie to konieczne.

Uwaga: w chwili wydania tej książki aktualną wersją Pythona 3 była wersja 3.6. W momencie gdy publikacja trafi w Twoje ręce, może się okazać, że najnowsza dostępna wersja będzie już miała wyższy numer, nie powinno Cię to jednak szczególnie martwić. Po prostu pobierz i zainstaluj wersję Pythona najbardziej aktualną w danej chwili.

Numer instalowanej przez Ciebie wersji najprawdopodobniej będzie się różnił od pokazanego tutaj. Nie przejmuj się, ponieważ sposób instalacji najnowszej wersji będzie taki sam.

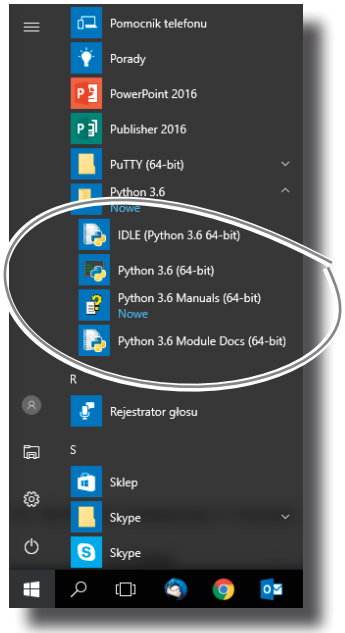


To naprawdę ważne: pamiętaj, aby zaznaczyć to pole wyboru przed kliknięciem pozycji Install Now (zainstaluj teraz) widocznej w tym oknie.

Sprawdź Pythona 3 w systemie Windows

Teraz, gdy interpreter Pythona jest już zainstalowany na Twojej maszynie działającej pod kontrolą systemu *Windows*, przeprowadźmy kilka prób, aby potwierdzić, że wszystko jest w porządku.

Na początek spójrz do swojego menu *Start*, w którym powinna się pojawić nowa grupa programów. Obok pokazaliśmy, jak wygląda to w systemie *Windows 10* zainstalowanym na jednej z maszyn w naszym laboratorium *Rusz głową!*. Powinieneś zaobserwować to samo na swoim komputerze. Jeśli tak nie jest, być może będziesz musiał powtórzyć proces instalacji. W starszych systemach operacyjnych, takich jak *Windows 7* czy *Windows 8*, również powinna się pojawić taka grupa, choć menu *Start* może w nich wyglądać nieco inaczej.



Instalator Pythona dodaje nową grupę do listy programów znajdującej się w systemowym menu Start.

Przejrzyjmy poszczególne elementy należące do grupy *Python 3.6*, zaczynając od dołu.

Pozycja *Python 3.6 Module Docs* (dokumentacja modułów Pythona 3.6) zapewnia dostęp do całej dokumentacji dostarczanej wraz ze wszystkimi zainstalowanymi modułami, które Twój system Python ma do dyspozycji. O modułach dowiesz się bardzo dużo w trakcie lektury tej książki, dlatego nie musisz w tej chwili martwić się tą pozycją ani niczego z nią robić.

Pozycja *Python 3.6 Manuals* (podręcznik Pythona 3.6) umożliwia otwarcie całego zbioru dokumentów dotyczących języka Python w standardowym narzędziu pomocy systemu *Windows*. Zawarty w nich materiał stanowi kopię dokumentacji Pythona 3 dostępnej w sieci.

Pozycja *Python 3.6* pozwala uruchomić interaktywny wiersz poleceń powłoki Pythona, za pomocą którego możesz na bieżąco eksperymentować z kodem wpisywanym po znaku zachęty `>>>`. Więcej informacji na temat wiersza poleceń powłoki Pythona znajdziesz już w rozdziale 1. Jeśli kliknąłeś tę pozycję, aby sprawdzić, co się stanie, i nie bardzo wiesz, co dalej robić, wpisz polecenie `quit()`, aby zamknąć okno powłoki i powrócić do systemu *Windows*.

Ostatnia omawiana przez nas pozycja (choć pierwsza na liście) to *IDLE (Python 3.6)*. Kliknięcie jej powoduje uruchomienie zintegrowanego środowiska programistycznego Pythona, noszącego nazwę *IDLE*. To bardzo proste IDE, które zapewnia dostęp do wiersza poleceń powłoki Pythona, całkiem przyzwoitego edytora tekstowego, debuggera Pythona oraz dokumentacji języka. Z *IDLE* będziemy korzystać bardzo często, a zaczniemy już w rozdziale 1.

To Python 3 działający pod kontrolą systemu Windows, tak jakby...

Python ma swoje korzenie w systemie operacyjnym *Unix* i środowiskach uniksopodobnych, co może czasami dać o sobie znać, gdy będziesz korzystał z tego języka w systemie *Windows*. Python zakłada na przykład, że pewne elementy oprogramowania powinny być obecne, jednak nie są one zawsze standardowo dostępne na platformie *Windows*, a zatem aby móc korzystać z pełni możliwości tego języka, programiści korzystający z *Windowsa* muszą często instalować niektóre rzeczy samodzielnie. Poświęćmy chwilę na zainstalowanie takiego ponadplanowego składnika, aby przedstawić sposób umożliwiający dodanie go, gdy tylko zajdzie taka potrzeba.

Dodawanie modułów do Pythona 3 w systemie Windows

Programiści używający Pythona działającego pod kontrolą systemu *Windows* czują się czasami oszukani, ponieważ niektórych możliwości, jakie są (zdaniem Pythona) dostępne na innych platformach, brakuje w środowisku *Windows*.

Na szczęście przedsiębiorczy programiści napisali niezależne moduły, które można zainstalować *w* Pythonie, zapewniając w ten sposób brakujące możliwości. Instalacja dowolnego z tych modułów wymaga tylko trochę pracy w wierszu poleceń systemu *Windows*.

Dodajmy na przykład pythonową implementację popularnej biblioteki `readline` do Twojej działającej w systemie *Windows* wersji Pythona. Dostęp do tej biblioteki zapewnia moduł `pyreadline`, który skutecznie łata dziurę polegającą na braku tej biblioteki w systemie *Windows*, występującą w każdej standardowej instalacji środowiska.

Otwórz okno wiersza poleceń systemu *Windows* i rób to, co my. Zamierzamy tu skorzystać z programistycznego narzędzia do instalowania (dołączonego do Pythona 3.6), aby zainstalować moduł `pyreadline`. Narzędzie to nosi nazwę `pip`, stanowiącą skrót od angielskich słów *Python Package Index*, oznaczających „indeks pakietów Pythona" i określających projekt, w wyniku którego powstało to narzędzie.

W wierszu poleceń systemu *Windows* wprowadź polecenie **`pip install pyreadline`**.

Porady maniaka

Biblioteka `readline` zapewnia implementację zbioru funkcji dających interaktywne możliwości edycji (używanych zwykle w wierszach poleceń). Moduł `pyreadline` zapewnia z kolei interfejs Pythona umożliwiający korzystanie z tej biblioteki.

To musisz wprowadzić w wierszu poleceń systemu Windows.

Wiersz polecenia

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\lsuma>
C:\Users\lsuma>pip install pyreadline
Collecting pyreadline
  Downloading pyreadline-2.1.zip (109kB)
    100% |                                | 112kB 134kB/s
Installing collected packages: pyreadline
  Running setup.py install for pyreadline ... done
Successfully installed pyreadline-2.1

C:\Users\lsuma>_
```

Tutaj zobaczysz kilka komunikatów.

Jeśli widzisz ten komunikat, wszystko jest w porządku.

Pamiętaj, że musisz połączyć się z internetem, zanim wydasz to polecenie.

Po zrobieniu tego moduł `pyreadline` jest zainstalowany i gotów do pracy w systemie *Windows*.

Teraz możesz przejść do rozdziału 1. i zacząć zapoznawać się z prostymi przykładami kodu Pythona.

Zainstaluj Pythona 3 w systemie Mac OS X (macOS)

Python 2 jest standardowo preinstalowany w systemie *Mac OS X*. Nie będziemy jednak mieli z tego żadnego pożytku, ponieważ zamiast wersji 2 chcemy używać Pythona 3. Na szczęście, gdy odwiedzasz serwis WWW poświęcony Pythonowi (*https://www.python.org/*), strona ta jest wystarczająca sprytna, aby odkryć, że korzystasz z komputera Mac. Umieść wskaźnik myszy nad kartą *Downloads*, po czym kliknij przycisk, aby pobrać instalator Pythona 3.6.x przeznaczony dla systemu *Mac OS X*. Wybierz najnowszą wersję Pythona 3, pobierz jego pakiet, a następnie zainstaluj go w sposób typowy dla Maca.

Program instalacyjny Pythona 3.6.1 standardowy dla systemu Mac OS X. Jeśli masz nowsze wydanie niż to, które jest tu widoczne, to świetnie — śmiało je zainstaluj!

Po prostu klikaj ten przycisk, aż Python zostanie zainstalowany.

Używanie menedżera pakietów

Na komputerach Mac możliwe jest również użycie jednego z popularnych opensource'owych *menedżerów pakietów*, takich jak *Homebrew* lub *MacPorts*. Jeśli nigdy nie korzystałeś z żadnego z nich, możesz swobodnie pominąć ten krótki punkt i od razu przeskoczyć do początku następnej strony. Jeśli jednak miałeś już do czynienia z którymkolwiek z nich, poniżej znajdziesz polecenia, za pomocą których możesz zainstalować Pythona 3 na swoim Macu z poziomu okna terminala:

- w przypadku narzędzia *Homebrew* wpisz polecenie **`brew install python3`**;
- w przypadku narzędzia *MacPorts* wpisz polecenie **`port install python3`**.

I to już wszystko — jesteś w domu. Python 3 jest gotów do działania w systemie *Mac OS X*. Zobaczmy, co zostało zainstalowane.

Sprawdź i skonfiguruj Pythona 3 w systemie Mac OS X

Aby przekonać się, czy instalacja w systemie *Mac OS X* się powiodła, kliknij ikonę *Programy* widoczną w Twoim doku, a następnie poszukaj folderu *Python 3.6*.

Kliknij ikonę folderu *Python 3.6*, a zobaczysz kilka ikon pokazanych poniżej.

Folder Python 3.6 w systemie Mac OS X

Pierwsza ikona, *IDLE*, jest zdecydowanie najbardziej przydatna, i to za jej pośrednictwem uruchamia się program, za pomocą którego przez większość czasu będziesz korzystał z Pythona podczas nauki tego języka. Kliknięcie tej ikony powoduje otwarcie zintegrowanego środowiska programistycznego Pythona — o nazwie *IDLE*. To bardzo proste IDE oferuje dostęp do interaktywnego wiersza poleceń powłoki Pythona, całkiem przyzwoitego edytora tekstowego, debuggera Pythona oraz dokumentacji języka. Z *IDLE* będziemy w tej książce korzystali naprawdę intensywnie.

Plik *Python Documentation.html* zawiera lokalną kopię całej dokumentacji Pythona w formacie HTML, którą możesz przeglądać za pomocą swojej przeglądarki internetowej (bez konieczności łączenia się z siecią).

Program *Python Launcher* jest automatycznie uruchamiany przez system *Mac OS X* za każdym razem, gdy dwukrotnie klikniesz plik wykonywalny zawierający kod Pythona. Mimo że program ten może się czasem komuś do czegoś przydać, w laboratorium *Rusz głową!* bardzo rzadko z niego korzystamy, warto go jednak mieć na wypadek, gdyby był nam kiedyś do czegoś potrzebny.

Ostatni element, plik *Update Shell Profile.command*, umożliwia aktualizację plików konfiguracyjnych systemu *Mac OS X* w celu zapewnienia, że lokalizacja interpretera Pythona oraz związanych z nim narzędzi została prawidłowo dodana do ścieżki Twojego systemu operacyjnego. Możesz kliknąć teraz ikonę tego pliku, aby uruchomić odpowiednie polecenie, a następnie zapomnieć o używaniu go w przyszłości — raz zupełnie wystarczy.

Zawartość folderu Python 3.6 należącego do folderu Programy w systemie Mac OS X.

Kliknij widoczną w doku ikonę Programy, po czym wybierz folder Python 3.6.

Jesteś gotów do działania w systemie Mac OS X

Dysponując wymienionymi elementami, jesteś gotów do pracy w systemie *Mac OS X*.

Teraz możesz przejść do rozdziału 1. i zacząć jego lekturę.

Zainstaluj Pythona 3 w systemie Linux

Jeśli korzystasz z najnowszej dystrybucji swojego ulubionego systemu *Linux*, *naprawdę doskonałą* wiadomością jest to, że prawdopodobnie masz już zainstalowanego Pythona 2 *oraz* Pythona 3.

Przedstawimy Ci tu prosty sposób umożliwiający poproszenie interpretera Pythona o podanie numeru obecnie zainstalowanej wersji. Aby to zrobić, otwórz systemowy wiersz poleceń i wprowadź następujące polecenie:

```
$ python3 -V
3.6.1
```

Zachowaj ostrożność — to WIELKA LITERA V.

Ale fajnie! Nasz Linux ma zainstalowaną najnowszą wersję Pythona 3.

Jeśli po tym, jak wydasz to polecenie, system *Linux* będzie narzekał, że nie może znaleźć programu `python3`, będziesz musiał zainstalować jego kopię. Sposób przeprowadzenia tej operacji zależy od dystrybucji *Linuksa*, której używasz.

Jeśli Twój *Linux* działa na bazie popularnych dystrybucji *Debian* lub *Ubuntu* (tak jak ma to miejsce w przypadku systemu używanego w laboratorium *Rusz głową!*), możesz skorzystać z narzędzia `apt-get` w celu zainstalowania Pythona 3. Oto polecenie, które powinieneś zastosować:

```
$ sudo apt-get install python3 idle3
```

Jeśli używasz dystrybucji wykorzystującej rozwiązanie *yum* lub *rpm*, użyj polecenia odpowiedniego dla tych systemów. Możesz też uruchomić swoje ulubione GUI *Linuksa* i skorzystać z oferowanego przez dystrybucję menedżera pakietów z interfejsem graficznym, aby wybrać do instalacji pakiety `python3` *oraz* `idle3`. W przypadku wielu systemów *Linux* popularnym rozwiązaniem jest tu *Synaptic Package Manager*, lecz istnieją również inne programy tego rodzaju, z których można skorzystać.

Pamiętaj, aby do instalacji w systemie Linux wybrać pakiety python3 oraz idle3.

Po zainstalowaniu Pythona 3 użyj polecenia przedstawionego na początku tej strony, aby sprawdzić, czy wszystko jest w porządku.

Niezależnie od tego, której dystrybucji używasz, polecenie `python3` umożliwia Ci uruchomienie interpretera Pythona w wierszu poleceń, podczas gdy polecenie `idle3` zapewnia dostęp do graficznego zintegrowanego środowiska programistycznego o nazwie *IDLE*. To bardzo proste IDE, które oferuje dostęp do interaktywnego wiersza poleceń powłoki Pythona, całkiem przyzwoitego edytora tekstowego, debuggera Pythona oraz dokumentacji języka.

Wiersza poleceń powłoki Pythona oraz środowiska *IDLE* będziemy bardzo często używać w tej książce, zaczynając już od rozdziału 1., do którego możesz teraz przejść.

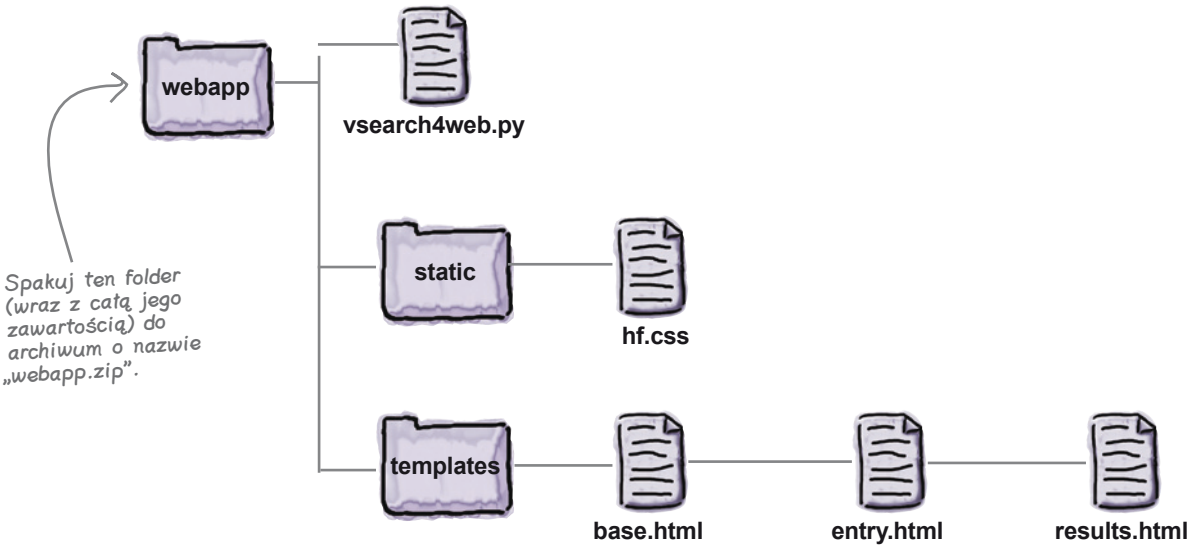
Wdrażanie Twojej aplikacji WWW

Na końcu rozdziału 5. stwierdziliśmy, że na wdrożenie Twojej aplikacji WWW w chmurze potrzebnych nam było zaledwie 10 minut.

Nadszedł czas na spełnienie tej obietnicy. W tym dodatku zamierzamy przeprowadzić Cię przez procedurę wdrażania Twojej aplikacji WWW w chmurze *PythonAnywhere*, zaczynając od zera, a kończąc na pełnym wdrożeniu w około 10 minut. *PythonAnywhere* to najbardziej popularna usługa w środowisku programistów Pythona i trudno nie zauważyć, dlaczego tak właśnie jest. Rozwiązanie to działa zgodnie z oczekiwaniami, oferuje doskonałe wsparcie dla języka Python (i frameworka Flask) oraz — co najlepsze — możesz zacząć udostępniać za jego pomocą swoje aplikacje zupełnie za darmo. Sprawdźmy, co możemy zdziałać z usługą *PythonAnywhere*.

Krok 0. Niewielkie przygotowania

W tej chwili kod naszej aplikacji WWW jest zapisany na dysku Twojego komputera w pliku *vsearch4web.py* znajdującym się w folderze o nazwie *webapp*, który oprócz tego pliku zawiera również foldery *static* oraz *templates* (jak zostało to pokazane poniżej). Aby przygotować aplikację do wdrożenia, utwórz archiwum ZIP zawierające wszystkie elementy znajdujące się w folderze *webapp*, nadając plikowi archiwum nazwę *webapp.zip*.



Oprócz pliku *webapp.zip* będziesz też musiał wgrać i zainstalować moduł `vsearch` przedstawiony w rozdziale 4. Na tym etapie musisz jedynie znaleźć plik dystrybucji, który wtedy utworzyłeś. Na naszym komputerze działającym pod kontrolą systemu operacyjnego *Windows* odpowiednie archiwum nosi nazwę *vsearch-1.0.tar.gz* i jest umieszczone w folderze *mymodules\vsearch\dist*. U Ciebie powinno być podobnie (również wtedy, gdy korzystasz z systemu operacyjnego *Mac OS X*).

Przypomnij sobie z rozdziału 4., że moduł setuptools języka Python umożliwia tworzenie archiwów tar.gz w każdym systemie operacyjnym.

Na razie nie musisz nic robić w żadnym z tych archiwów. Po prostu zanotuj sobie, gdzie na Twoim komputerze się one znajdują, abyś mógł je łatwo znaleźć, gdy będziesz je wgrywał na serwer usługi *PythonAnywhere*. Śmiało, weź ołówek i nagryzmol sobie poniżej informacje na temat lokalizacji obydwu plików.

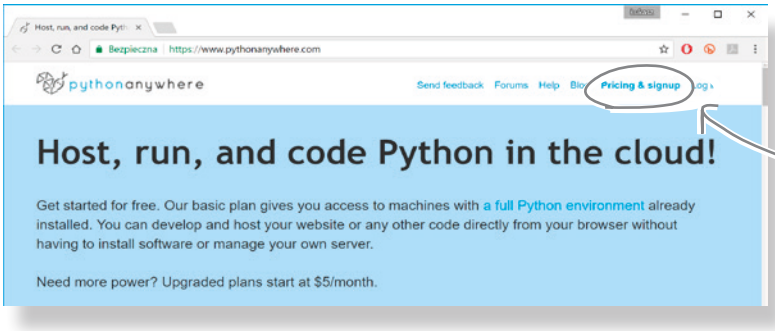
`webapp.zip` _____

`vsearch-1.0.tar.gz` _____

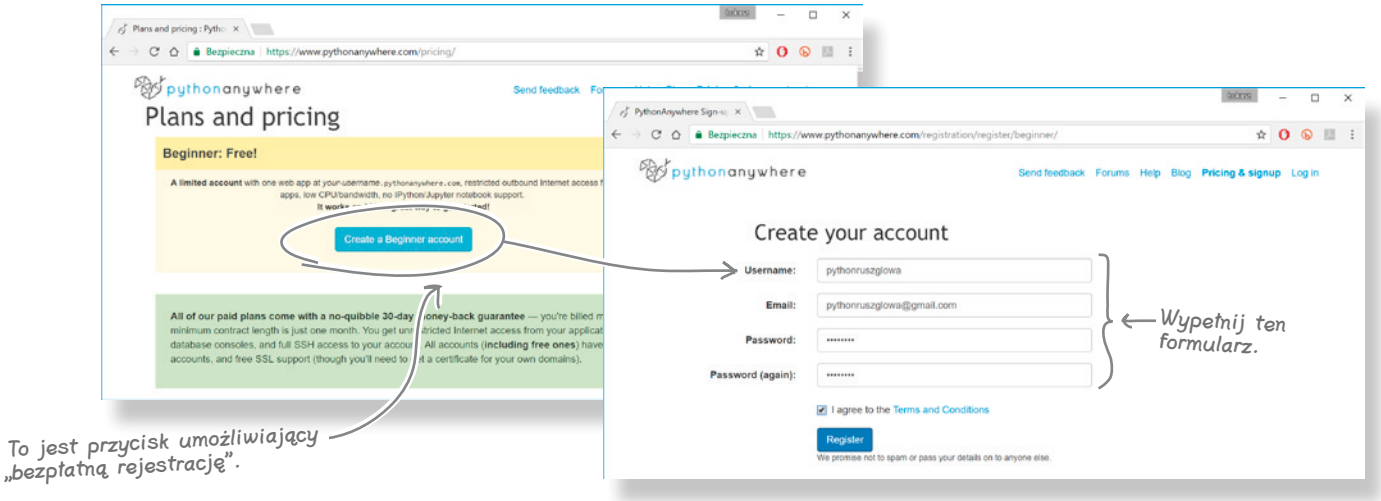
This is "vsearch.zip" instead if you're on Windows.

Krok 1. Zarejestruj się w serwisie PythonAnywhere

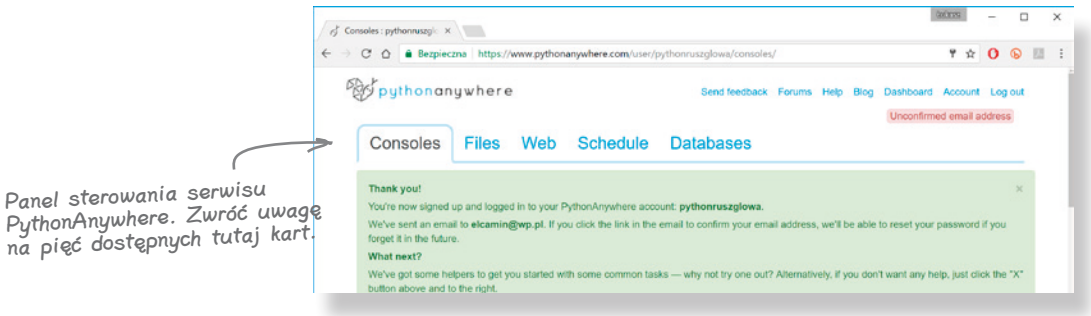
Ten krok nie mógłby być prostszy. Przejdź na stronę *https://www.pythonanywhere.com*, po czym kliknij widoczne na niej łącze *Pricing & signup* (ceny i rejestracja).



Kliknij wielki niebieski przycisk *Create a Beginner account* (utwórz konto początkującego użytkownika), aby utworzyć nowe konto, po czym wypełnij formularz rejestracyjny odpowiednimi danymi.

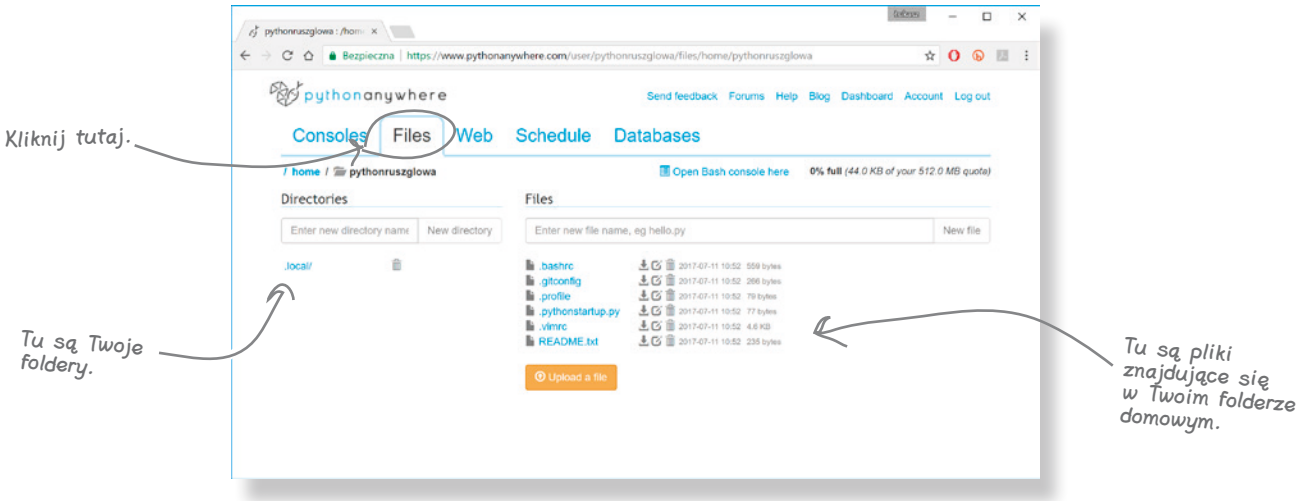


Jeśli wszystko pójdzie dobrze, w oknie przeglądarki ujrzysz panel sterowania serwisu *PythonAnywhere*. Zwróć uwagę na to, że w tym momencie jesteś już zarejestrowany *oraz* zalogowany w serwisie.

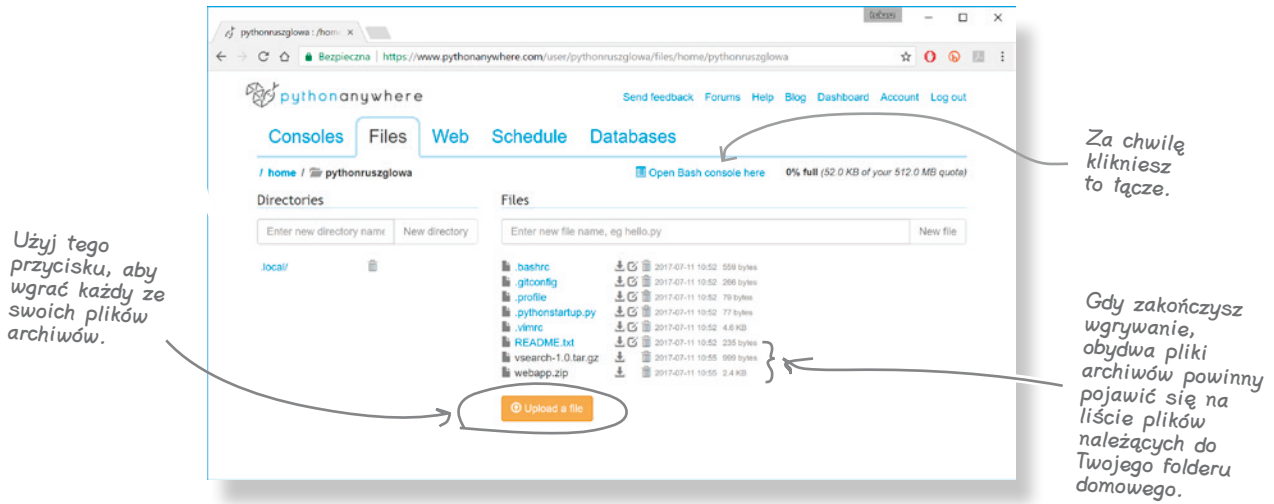


Krok 2. Wgraj swoje pliki do chmury

Kliknij kartę *Files* (pliki), aby wyświetlić dostępne foldery i pliki.



Użyj przycisku *Upload a file* (wgraj plik), aby znaleźć i wgrać dwa pliki archiwów, o których była mowa w **kroku 0**. Utwórz folder, który będzie zawierał pliki Twojej strony. Wpisz nazwę '*mysite*' w polu tekstowym *Enter new directory name* i kliknij przycisk *New directory*.



Jesteś teraz gotów do rozpakowania i zainstalowania dwóch wgranych przed chwilą archiwów; zrobisz to w **kroku 3**. Zacznij od kliknięcia łącza *Open Bash console here* (otwórz tutaj konsolę Bash) znajdującego się w okolicach prawego górnego narożnika przedstawionej powyżej strony. Spowoduje to otwarcie okna terminala w Twojej przeglądarce internetowej (za pomocą której odwiedziłeś serwis *PythonAnywhere*).

Krok 3. Rozpakuj i zainstaluj swój kod

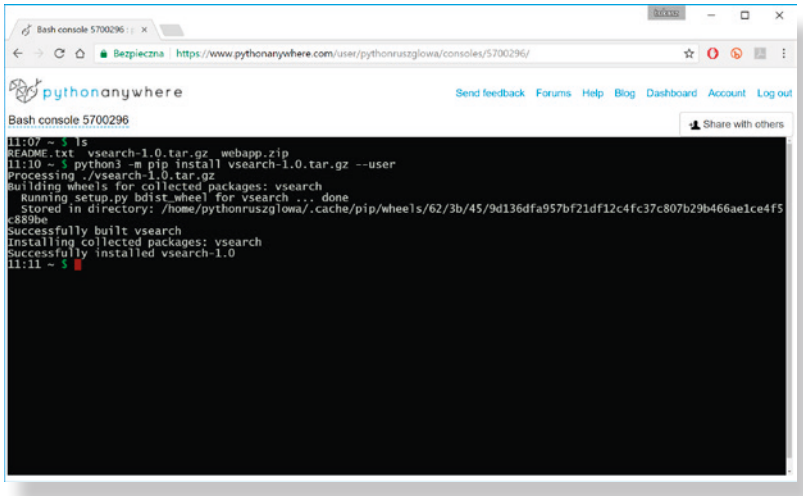
Gdy klikniesz łącze *Open Bash console here*, serwis *PythonAnywhere* odpowie, zastępując panel sterowania z kartą *Files* działającą w oknie przeglądarki konsolą (wierszem poleceń) *Linuksa*. Za jej pomocą wydasz kilka poleceń, których wykonanie spowoduje rozpakowanie i zainstalowanie modułu `vsearch` oraz kodu Twojej aplikacji WWW. Zacznij od zainstalowania modułu `vsearch` w Pythonie jako „prywatnego modułu" (tj. tylko do Twojego użytku), korzystając z przedstawionego poniżej polecenia.

```
python3 -m pip install vsearch-1.0.tar.gz --user
```

Dzięki zastosowaniu parametru `--user` moduł `vsearch` zostanie zainstalowany tylko do Twojego użytku. Serwis PythonAnywhere nie umożliwia Ci instalowania modułów do użytku wszystkich (lecz wyłącznie do Twojego).

Uruchom to polecenie.

Sukces!



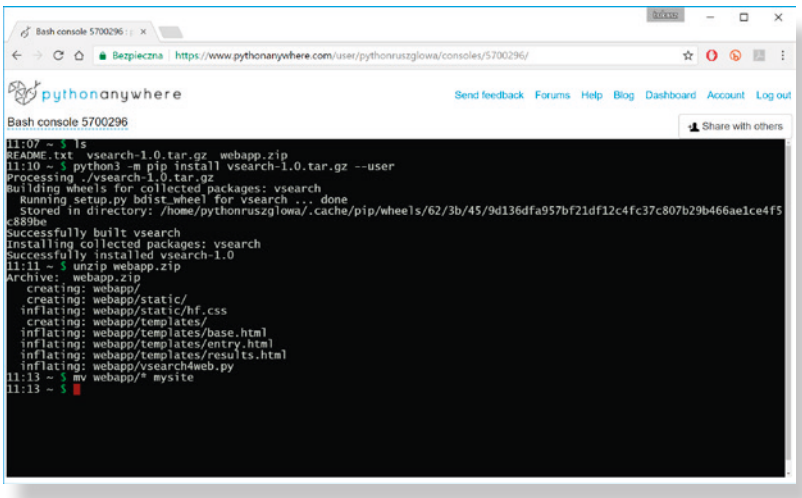
Po udanym zainstalowaniu modułu `vsearch` nadszedł czas na zwrócenie uwagi na kod Twojej aplikacji WWW, który powinien zostać zainstalowany w folderze *mysite* (istniejącym już w Twoim folderze domowym w serwisie *PythonAnywhere*). Aby to zrobić, wydaj dwa przedstawione poniżej polecenia.

Rozpakuj kod swojej aplikacji WWW...

```
unzip webapp.zip
mv webapp/* mysite
```

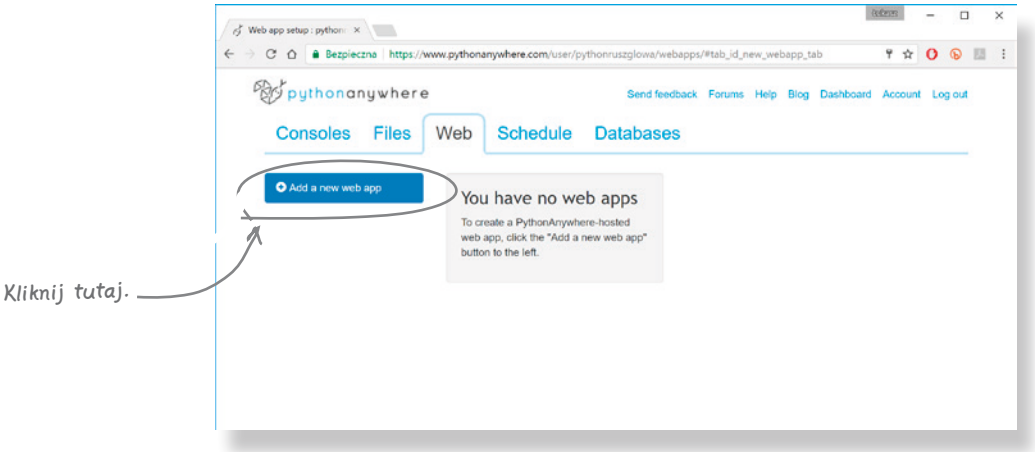
...a następnie przenieś go do folderu mysite.

Powinieneś zobaczyć komunikaty podobne do przedstawionych tutaj.

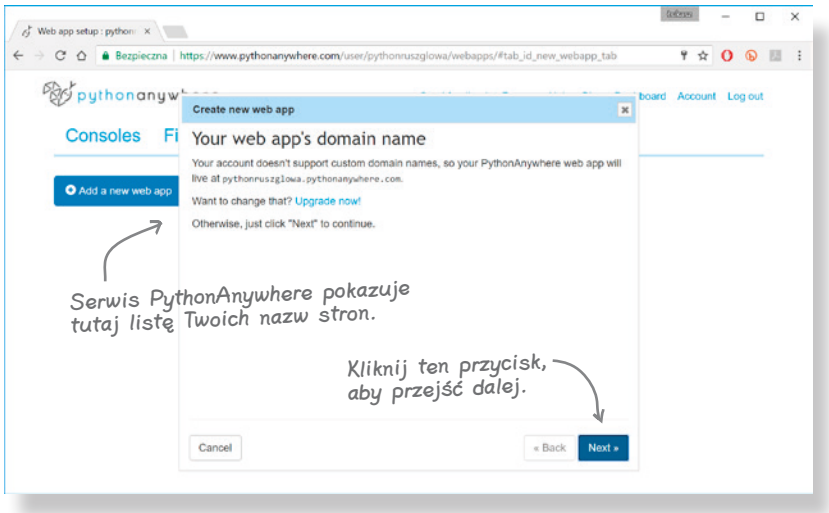


Krok 4. Utwórz starter aplikacji WWW (1. z 2)

Po wykonaniu **kroku 3.** powróć do panelu sterowania serwisu *PythonAnywhere* i wybierz zakładkę *Web* (WWW), w której możesz utworzyć nowy starter aplikacji WWW. Po zrobieniu tego będziesz mógł zastąpić domyślny kod startera własnym kodem. Zwróć uwagę, że każde konto początkującego użytkownika umożliwia udostępnienie za darmo tylko jednej aplikacji WWW. Jeśli chcesz mieć ich więcej, będziesz musiał skorzystać z płatnego konta. Na szczęście — w tej chwili — potrzebna nam jest tylko jedna aplikacja, więc przejdźmy dalej, klikając niebieski przycisk *Add a new web app* (dodaj nową aplikację WWW).



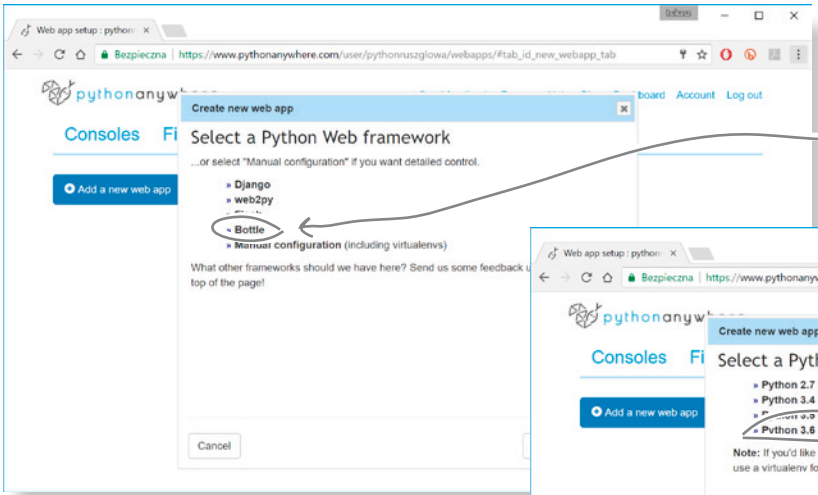
Jako że korzystasz z bezpłatnego konta, Twoja aplikacja WWW będzie dostępna pod adresem pokazanym na następnym ekranie kreatora. Kliknij przycisk *Next* (dalej), aby potwierdzić nazwę strony zasugerowaną przez serwis *PythonAnywhere* i przejść dalej.



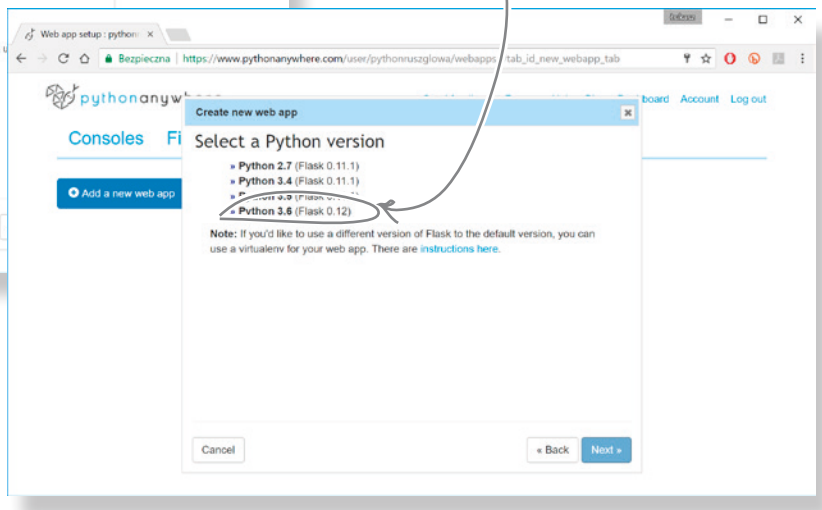
Kliknij przycisk *Next*, aby kontynuować.

Krok 4. Utwórz starter aplikacji WWW (2. z 2)

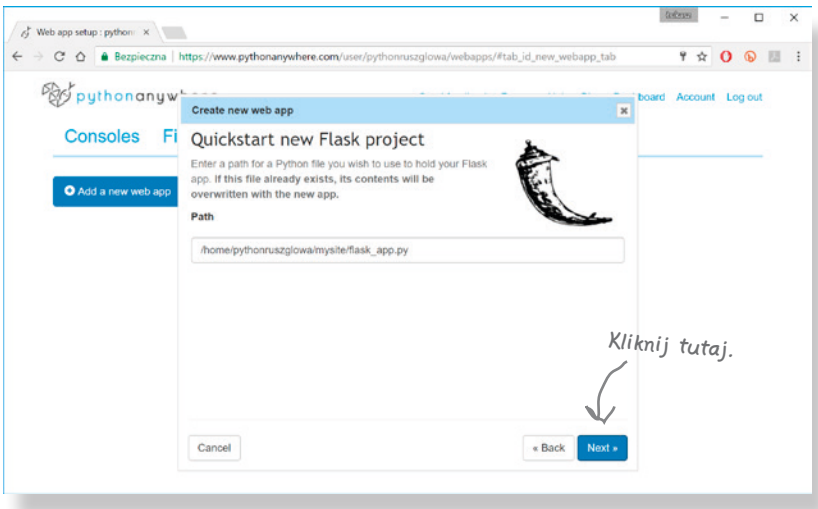
Serwis *PythonAnywhere* zapewnia obsługę wielu frameworków sieciowych, dlatego na następnym ekranie kreatora możesz wybrać spośród wielu dostępnych tu systemów. Wybierz framework Flask, a następnie wybierz wersje Flaska i Pythona, przy użyciu których chcesz wdrożyć swoją aplikację. W czasie pisania tej książki najaktualniejszymi wersjami obsługiwanymi przez serwis *PythonAnywhere* były Python 3.6 oraz Flask 0.12, wybierz więc tę kombinację, chyba że oferowana jest nowsza (wtedy powinieneś oczywiście wybrać tę nowszą).



Dla swojej aplikacji WWW wybierz framework Flask, po czym wybierz najbardziej aktualną kombinację Pythona i Flaska.



Jesteś już niemal u celu. Kolejny ekran kreatora umożliwia utworzenie na początek prostej aplikacji WWW wykorzystującej framework Flask. Zrób to teraz, akceptując wartości pokazane na tej stronie i klikając przycisk *Next*, aby kontynuować.

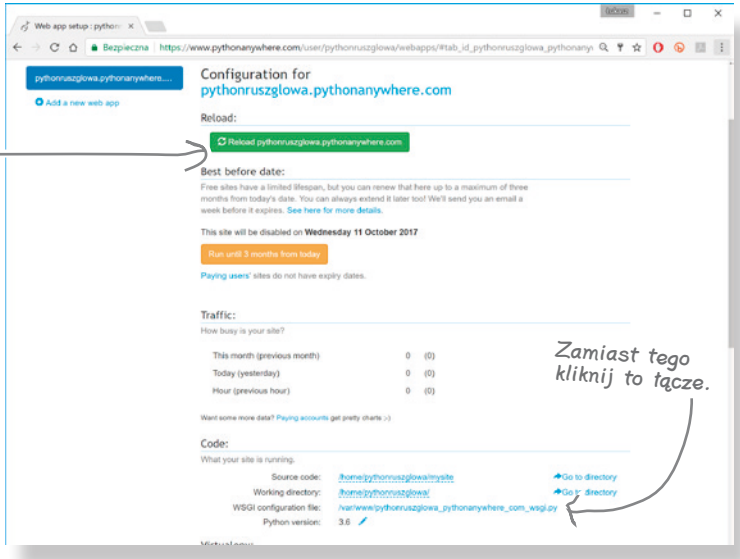


Nie musisz tu klikać przycisku Next. Gdy tylko wybierzesz odpowiednią kombinację, pojawi się ten ekran.

Kliknij tutaj.

Krok 5. Skonfiguruj swoją aplikację WWW

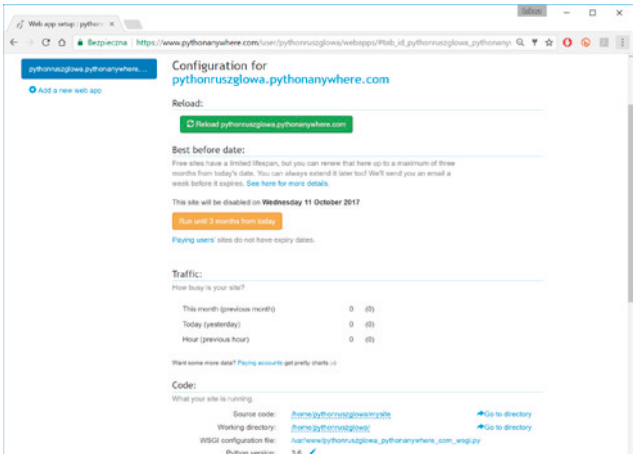
Po wykonaniu **kroku 4.** ujrzysz zakładkę *Web* (WWW) panelu sterowania. Nie ulegaj na tym etapie pokusie kliknięcia wielkiego zielonego przycisku, ponieważ nie poinformowałeś jeszcze serwisu *PythonAnywhere* o swoim kodzie; wstrzymaj się zatem przez chwilę z uruchamianiem czegokolwiek. Zamiast tego kliknij długie łącze widoczne na prawo od etykiety *WSGI configuration file* (plik konfiguracyjny WSGI).



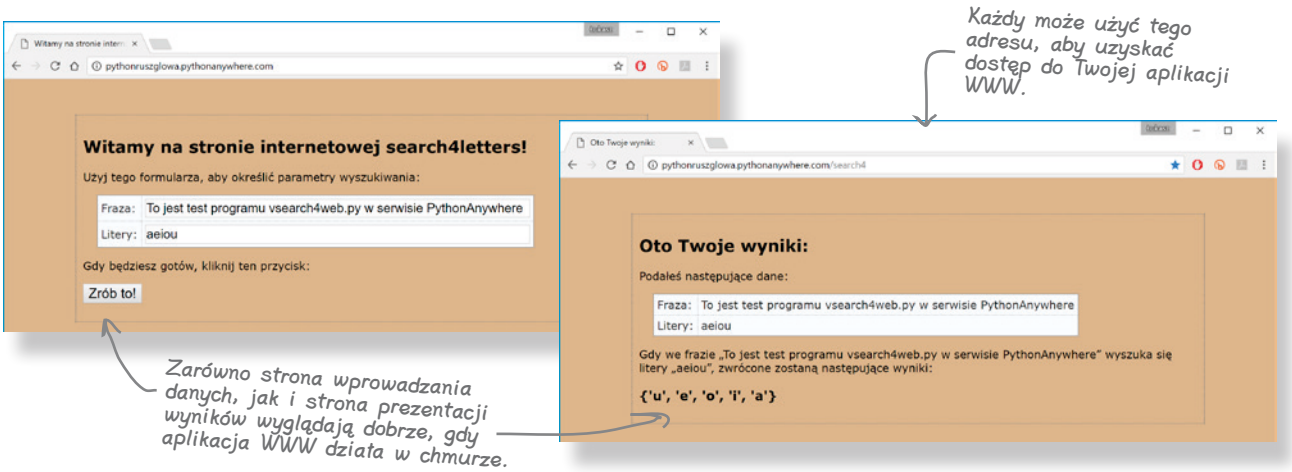
Kliknięcie tego długiego łącza spowoduje, że do działającego w przeglądarce internetowej edytora tekstowego serwisu *PythonAnywhere* załadowany zostanie plik konfiguracyjny Twojej nowo utworzonej aplikacji WWW wykorzystującej framework Flask. Na końcu rozdziału 5. poinformowaliśmy Cię, że serwis *PythonAnywhere* importuje Twoją aplikację WWW przed wywołaniem metody `app.run()` w Twoim imieniu. Obsługę tego mechanizmu zapewnia właśnie wspomniany tu plik konfiguracyjny. Trzeba jednak sprawić, aby odwoływał się on do *Twojego* kodu, nie zaś do kodu należącego do aplikacji startera, dlatego musisz zmienić ostatni wiersz tego pliku (tak jak zostało to pokazane poniżej), a następnie kliknąć przycisk *Save* (zapisz).

Krok 6. Weź w obroty swoją aplikację WWW działającą w chmurze!

Pamiętaj, aby zapisać swój zmieniony plik konfiguracyjny, a następnie powróć do zakładki *Web* panelu sterowania. Nadszedł czas na kliknięcie tego wielkiego kuszącego zielonego przycisku. Zrób to teraz!



Po krótkiej chwili serwis przeładuje konfigurację, a Ty kliknij łącze widoczne ponad tym przyciskiem w przeglądarce pojawi się strona Twojej aplikacji WWW, która będzie działać dokładnie tak samo, jak miało to miejsce, gdy była uruchamiana lokalnie, lecz teraz może z niej korzystać każdy, kto ma dostęp do internetu i przeglądarki internetowej.



W tym momencie jesteś już w domu. Aplikacja WWW, którą opracowaliśmy w rozdziale 5., została wdrożona w chmurze *PythonAnywhere* (w mniej niż 10 minut). Serwis ten oferuje Ci oczywiście znacznie więcej możliwości, niż udało nam się zaprezentować w tym krótkim dodatku, więc nie krępuj się, jeśli masz ochotę dowiedzieć się o nich czegoś więcej i sprawdzić je w praktyce. Pamiętaj, aby w odpowiednim momencie powrócić do panelu sterującego serwisu *PythonAnywhere* i wylogować się. Zauważ, że po tym, jak to zrobisz, Twoja aplikacja WWW nadal będzie działać w chmurze, dopóki nie każesz jej przestać (lub dopóki nie skończy się trzymiesięczny okres ważności witryny działającej na bezpłatnym koncie, który możesz jednak przedłużać). Nieźle, nie?

Zawsze można się czegoś jeszcze nauczyć

Nigdy nie było naszą intencją opisanie wszystkich tematów.

Celem tej książki było od samego początku przedstawienie takich informacji na temat Pythona, które pozwolą Ci jak najszybciej zacząć praktycznie posługiwać się tym językiem. Istnieje znacznie więcej zagadnień, które moglibyśmy tu poruszyć, ale tego nie zrobiliśmy. W tym dodatku omawiamy 10 najważniejszych rzeczy, którymi — gdybyśmy mieli kolejnych 600 (lub coś koło tego) stron do dyspozycji — zajęlibyśmy się również. Nie wszystkie z tych tematów Cię zainteresują, mimo to szybko je przejrzyj na wypadek, gdybyśmy przypadkiem trafili w jakiś Twój czuły punkt lub odpowiedzieli na dręczące Cię pytanie. Wszystkie opisane w tym dodatku rozwiązania programistyczne są z definicji dostępne w języku Python i jego interpreterze.

1. Co z Pythonem 2?

W momencie publikacji tej książki (czyli w trzecim kwartale 2017 roku) w powszechnym użyciu były dwie główne gałęzie Pythona. Wiesz już całkiem sporo o języku **Python 3**, ponieważ właśnie jego używaliśmy w tej książce.

Wszystkie nowe możliwości i rozszerzenia są wprowadzane w Pythonie 3, w przypadku którego cykl wydawania mniejszych wersji trwa od 12 do 18 miesięcy. Wydanie kolejnej wersji tego typu, oznaczonej numerem 3.7, planowane jest na połowę 2018 roku; data planowanego wydania wersji 3.8 nie jest jeszcze znana.

Pyton 2 „utknął" jakiś czas temu na wersji 2.7. Ma to związek z faktem, że *programiści rdzenia Pythona* (czyli ludzie przewodzący rozwojowi tego języka) postanowili, że przyszłość należy do wersji 3, natomiast wersja 2 powinna po cichu odejść w przeszłość. Decyzja ta została umotywowana solidnymi argumentami technicznymi, jednak nikt tak naprawdę nie spodziewał się, że obecna sytuacja potrwa tak długo. Jakby nie było, Python 3 — mający być przyszłością języka — pojawił się na rynku pod koniec 2008 roku.

Na temat tego, co się stało od końca 2008 roku do teraz, można by było napisać całą książkę. Wystarczy jednak stwierdzić, że Python 2 uparcie odmawia od tej pory odejścia w przeszłość. W momencie wydania Pythona 3 istniała ogromna baza zainstalowanego kodu Pythona 2 (która istnieje zresztą do dziś), a także istniało bardzo wielu używających go programistów, którzy w pewnych obszarach zastosowań mocno ociągali się z aktualizacją swojego kodu. Jest ku temu bardzo prosty powód: Python 3 wprowadził sporo przydatnych rozszerzeń, które zniszczyły wsteczną kompatybilność. Inaczej mówiąc: istnieje mnóstwo kodu napisanego w Pythonie 2, który to kod nie będzie działał *bez zmian* w Pythonie 3 (mimo że na pierwszy rzut oka trudno będzie rozróżnić kod Pythona 2 od kodu Pythona 3). Wielu programistów wierzyło również, że Python 2 jest „wystarczająco dobry" i po prostu z tego powodu nie zaktualizowało swojego kodu.

Ostatnio (to jest w minionych dwóch latach) następuje całkowita przemiana. Liczba osób przesiadających się z wersji 2 na 3 wydaje się rosnąć. Niektóre bardzo popularne moduły niezależne zostały wydane w wersjach zgodnych z Pythonem 3, co ma pozytywny wpływ na przyjęcie Pythona 3. Ponadto *programiści rdzenia Pythona* bez przerwy dodają nowe możliwości do tej wersji języka, sprawiając, że z czasem staje się ona coraz bardziej atrakcyjną alternatywą. Praktyka „przenoszenia" interesujących nowych możliwości z wersji 3 do 2 została porzucona w wersji 2.7 i chociaż poprawki błędów i łatki zabezpieczeń są nadal wydawane, *programiści rdzenia Pythona* zapowiedzieli już, że również ta działalność zostanie zakończona, a stanie się to w 2020 roku. Dla Pythona 2 zegar nieubłaganie tyka.

Oto ogólna porada dla wszystkich zastanawiających się, którą wersję Pythona wybrać:

Jeśli zaczynasz nowy projekt, użyj Pythona 3.

Powinieneś oprzeć się chęci tworzenia kodu „starego typu" w języku Python 2, zwłaszcza gdy zaczynasz zupełnie od zera. Jeśli musisz zajmować się jakimś istniejącym kodem Pythona 2, to, czego nauczyłeś się o Pythonie 3, będzie miało zastosowanie i z pewnością będziesz w stanie przeczytać kod i go zrozumieć (w końcu to nadal język Python, niezależnie od głównego numeru wersji). Jeśli istnieją jakieś techniczne powody, dla których kod powinien nadal działać w Pythonie 2, niech tak będzie. Jeśli jednak kod może bez większego zamieszania zostać przeniesiony do Pythona 3, to warto zadać sobie trochę trudu, aby to zrobić, ponieważ Python 3 *jest* lepszym językiem i jest przyszłością.

2. Wirtualne środowiska programistyczne

Wyobraź sobie, że masz dwóch klientów, z których jeden ma kod Pythona i jest uzależniony od jednej wersji pewnego niezależnego modułu, a drugi ma kod, który jest zależny od *innej* wersji tego samego modułu. Ty zaś jesteś oczywiście tym biedakiem odpowiedzialnym za utrzymanie kodu tworzącego obydwa projekty.

Robienie tego na jednym komputerze może być kłopotliwe, ponieważ interpreter Pythona nie umożliwia instalacji różnych wersji modułów niezależnych.

Istnieje jednak rozwiązanie tego problemu dzięki możliwości korzystania ze środowisk wirtualnych.

Środowisko wirtualne (ang. *virtual environment*) umożliwia tworzenie nowych, czystych środowisk Pythona, wewnątrz których możesz uruchamiać swój kod. Dzięki temu jesteś w stanie instalować niezależne moduły w jednym środowisku wirtualnym, nie wpływając tym samym na inne środowiska, a na swoim komputerze możesz korzystać z dowolnej liczby środowisk wirtualnych, przełączając się pomiędzy nimi przez *aktywowanie* tego, którego chcesz w danej chwili używać. Dzięki temu, że każde środowisko wirtualne może zawierać własną kopię dowolnych modułów niezależnych, które zechcesz zainstalować, możesz korzystać z dwóch różnych środowisk wirtualnych, po jednym dla każdego z klienckich projektów, o których była mowa powyżej.

Jestem prawie pewien, że udało mi się rozwiązać mój problem z wieloma wersjami niezależnego modułu... W tym celu musiałem tylko przeczytać wszystkie te książki.

Zanim jednak to zrobisz, musisz podjąć decyzję dotyczącą tego, której technologii środowisk wirtualnych należy użyć. Do dyspozycji masz w tej kwestii dwa rozwiązania: jedno, o nazwie `venv`, które jest dostarczane wraz z *biblioteką standardową* języka Python 3, oraz drugie, o nazwie `virtualenv`, będące modułem, który musisz pobrać z serwisu PyPI i zainstalować w swoim systemie (zapewnia ono te same możliwości, co `venv`, ale oferuje dodatkowe bajery). Najlepiej będzie, gdy dokonasz tu świadomego wyboru.

Aby lepiej poznać rozwiązanie `venv`, odwiedź stronę zawierającą jego dokumentację:

https://docs.python.org/3/library/venv.html.

Aby dowiedzieć się, co oferuje moduł `virtualenv` oprócz możliwości zapewnianych przez `venv`, zerknij tutaj:

https://pypi.org/project/virtualenv/.

To, czy na potrzeby swoich projektów będziesz korzystał ze środowisk wirtualnych, jest Twoją osobistą decyzją. Niektórzy programiści zarzekają się, że nie napiszą nawet jednego wiersza kodu Pythona, dopóki nie będą mogli uruchomić go w środowisku wirtualnym. Jest to być może dość skrajne stanowisko, ale każdy pracuje tak, jak mu wygodnie.

Postanowiliśmy nie opisywać środowisk wirtualnych w głównej części tej książki. Uważamy, że są one prawdziwym wybawieniem, gdy zachodzi potrzeba ich użycia, ale nie wierzymy, że każdy programista Pythona musi korzystać ze środowiska wirtualnego w przypadku wszystkiego, co robi.

Zalecamy Ci omijać szerokim łukiem ludzi, którzy twierdzą, że nie jesteś prawdziwym programistą Pythona, jeśli nie używasz rozwiązania `virtualenv`.

Wszystko, co musiał zrobić, to użyć wirtualnego środowiska.

3. Więcej na temat obiektowości

Jeśli przeczytałeś całą tę książkę, powinieneś już (a przynajmniej mamy taką nadzieję) zdawać sobie sprawę ze znaczenia zdania: „W języku Python wszystko jest obiektem”.

Oferowane przez Pythona możliwości zastosowania obiektów są wspaniałe. Zasadniczo oznacza to, że wszystko działa tu tak, jak się tego spodziewasz. Jednak z faktu, że wszystko jest obiektem, **nie** wynika, że wszystko musi należeć do jakiejś klasy, zwłaszcza jeśli chodzi o Twój kod.

W tej książce nie przedstawiliśmy Ci sposobu tworzenia własnej klasy aż do momentu, w którym naprawdę go potrzebowaliśmy, aby utworzyć własny menedżer kontekstu. Jednak nawet wtedy omówiliśmy tylko to, co było nam niezbędne, i nic innego. Jeśli przesiadasz się na Pythona z języka programowania, w którym cały Twój kod musi znajdować się w klasie (czego klasycznym przykładem jest *Java*), zastosowany tu przez nas sposób prezentacji treści może Ci się wydawać dość nienaturalny. Nie martw się tym jednak, ponieważ Python jest znacznie mniej rygorystyczny niż (na przykład) *Java*, jeśli chodzi o to, jak piszesz swoje programy.

Jeśli postanowisz utworzyć kilka funkcji wykonujących potrzebne Ci operacje, możesz to śmiało zrobić. Jeśli myślisz w bardziej funkcyjny sposób, Python również może Ci pomóc, oferując składnię komprehencji, stanowiącą ukłon w stronę programowania funkcyjnego. A jeśli nie możesz uciec od konieczności umieszczenia swojego kodu w klasie, Python zapewnia Ci pełną składnię obiektową wbudowaną w język.

Jeśli okaże się, że spędzasz mnóstwo czasu na tworzeniu klas, sprawdź następujące konstrukcje:

- `@staticmethod` — dekorator umożliwiający tworzenie w klasie funkcji statycznej (która nie otrzymuje `self` jako pierwszego argumentu);
- `@classmethod` — dekorator umożliwiający tworzenie metody klasy, która to metoda oczekuje klasy jako swojego pierwszego argumentu (noszącego zwykle nazwę `cls`), nie zaś argumentu `self`;
- `@property` — dekorator umożliwiający zmianę charakteru metody w taki sposób, aby mogła być używana, jakby była atrybutem;
- `__slots__` — dyrektywa klasy, która to dyrektywa (gdy się jej użyje) może znacznie poprawić wydajność pamięciową obiektów tworzonych z Twojej klasy (pewnym kosztem elastyczności).

Aby dowiedzieć się więcej na temat tych konstrukcji, zerknij do dokumentacji języka Python (*https://docs.python.org/3/*) albo sięgnij po którąś z naszych ulubionych książek na temat Pythona (opisanych w jednym z kolejnych dodatków).

4. Specyfikatory formatów łańcuchów znakowych i tym podobne

Aplikacja wykorzystywana wielokrotnie w tej książce jako przykład wyświetlała swoje dane wyjściowe w przeglądarce internetowej. Umożliwiło nam to pozostawienie kwestii formatowania kodowi HTML (a konkretnie skorzystaliśmy w tym celu z modułu *Jinja2* należącego do frameworka *Flask*). Co za tym idzie, pominęliśmy tu pewien obszar, w którym język Python błyszczy szczególnie mocno, a mianowicie zagadnienia tekstowego formatowania łańcuchów znakowych.

Powiedzmy, że masz łańcuch znakowy, który ma zawierać pewne wartości, lecz wartości te nie będą znane aż do czasu uruchomienia kodu. Chcesz utworzyć komunikat (zapisany w zmiennej `msg`) zawierający wspomniane wartości, aby móc później wykonać jakąś operację (na przykład wyświetlić komunikat na ekranie, włączyć go w treść strony HTML, którą utworzysz za pomocą silnika *Jinja2*, lub udostępnić trzem milionom swoich fanów za pomocą Twittera). Wartości generowane przez Twój kod w czasie wykonania znajdują się w dwóch zmiennych: `price` (wartość ta określa cenę danego towaru) oraz `tag` (wartość ta zawiera jakieś chwytliwe hasło reklamowe). Masz tu do wyboru kilka opcji. I tak możesz:

- zbudować potrzebny Ci komunikat, korzystając z konkatenacji (operatora +);
- użyć staromodnych specyfikatorów formatów łańcuchów znakowych (korzystając ze znaków %);
- wykorzystać dostępną dla każdego łańcucha znakowego metodę `format` w celu zbudowania komunikatu.

Poniżej znajdziesz krótką sesję wiersza poleceń, w której wszystkie te sposoby zostały zaprezentowane w praktyce (zakładamy, że po lekturze tej książki zgodzisz się z przesłaniem wygenerowanego komunikatu).

```
Python 3.6.1 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>> price = 99.99
>>> tag = 'to doskonała okazja!'
>>>
>>> msg = 'Książka "Python. Rusz głową!" za ' + str(price) + ' zł ' + tag
>>> msg
'Książka "Python. Rusz głową!" za 99.99 zł to doskonała okazja!'
>>>
>>> msg = 'Książka "Python. Rusz głową!" za %2.2f zł %s' % (price, tag)
>>> msg
'Książka "Python. Rusz głową!" za 99.99 zł to doskonała okazja!'
>>>
>>> msg = 'Książka "Python. Rusz głową!" za {} zł {}'.format(price, tag)
>>> msg
'Książka "Python. Rusz głową!" za 99.99 zł to doskonała okazja!'
>>>
                                                    Ln: 44  Col: 4
```

Wiedziałeś to już wcześniej, prawda? :)

Specyfikatory formatu %s i %f są stare jak świat, ale... co tam, podobnie jak ja, nadal świetnie działają.

Kwestią Twojego osobistego wyboru jest to, którego z przedstawionych tu sposobów użyjesz, chociaż istnieje presja na wspieranie stosowania metody `format` zamiast pozostałych dwóch rozwiązań (więcej informacji na ten temat znajdziesz w dokumencie *PEP 3101*, który jest dostępny pod adresem *https://www.python.org/dev/peps/pep-3101/*). W praktyce będziesz miał do czynienia z kodem, w którym wykorzystywany jest wyłącznie jeden sposób, jak również z takim, w którym pojawiają się wszystkie trzy rozwiązania (co niekoniecznie ułatwia pracę). Aby dowiedzieć się więcej na ten temat, odwiedź stronę:

https://docs.python.org/3/library/string.html#formatspec.

5. Sortowanie rzeczy

Python zapewnia doskonałe wbudowane możliwości w zakresie sortowania elementów. Niektóre wbudowane struktury danych tego języka (na przykład listy) oferują metody `sort`, z których można korzystać w celu porządkowania swoich danych znajdujących się w tych strukturach. Tym, co czyni Pythona naprawdę wyjątkowym, jest jednak wbudowana funkcja `sorted` (ponieważ BIF działa z *dowolną* z wbudowanych struktur danych).

BIF to skrót od angielskich słów „built-in function" oznaczających „funkcja wbudowana".

W przedstawionej poniżej sesji środowiska IDLE definiujemy najpierw niewielki słownik (o nazwie `product`), który następnie przetwarzamy za pomocą kolejnych pętli `for`. BIF `sorted` jest tu używana do kontrolowania porządku elementów słownika przekazywanego do każdej pętli `for`. Wykonaj poszczególne działania na swoim komputerze, czytając kolejne komentarze.

Python 3.6.1 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>> product = {'Książka': 99.99,
               'PDF': 89.99,
               'Wideo': 299.99}
>>>
>>> for k in product:
        print(k, '->', product[k])

PDF -> 89.99
Książka -> 99.99
Wideo -> 299.99
>>>
>>> for k in sorted(product):
        print(k, '->', product[k])

Książka -> 99.99
PDF -> 89.99
Wideo -> 299.99
>>>
>>> for k in sorted(product, key=product.get):
        print(k, '->', product[k])

PDF -> 89.99
Książka -> 99.99
Wideo -> 299.99
>>>
>>> for k in sorted(product, key=product.get, reverse=True):
        print(k, '->', product[k])

Wideo -> 299.99
Książka -> 99.99
PDF -> 89.99
>>>
```

Ln: 82 Col: 4

Zdefiniuj słownik (pamiętaj, że kolejność wstawiania elementów NIE jest zachowywana).

Wyświetl słownik na ekranie.

Surowe dane wydają się posortowane według wartości, lecz tak naprawdę nie są (to po prostu kwestia szczęścia, że tak się ułożyły).

Dodanie wywołania metody sorted powoduje uporządkowanie słownika według klucza, co może, lecz nie musi, być tym, o co Ci tu chodzi.

Litera „K" występuje przed literą „P", która występuje z kolei przed literą „W". Sortowanie według kluczy (będące standardowym trybem działania funkcji) działa prawidłowo.

Dodanie argumentu key powoduje sortowanie według wartości.

Dane wyjściowe są uporządkowane według ceny (od najniższej do najwyższej).

Dane wyjściowe są teraz uporządkowane według ceny, lecz w kolejności odwrotnej (od najwyższej do najniższej).

Dodanie argumentu reverse odwraca porządek sortowania.

Więcej informacji na temat sortowania w języku Python znajdziesz w doskonałym materiale znajdującym się tutaj: *https://docs.python.org/3/howto/sorting.html#sortinghowto*.

6. Więcej z biblioteki standardowej

Biblioteka standardowa Pythona pełna jest wszelkiego rodzaju dobra. Warto raz na jakiś czas wykonać ćwiczenie polegające na poświęceniu 20 minut na przegląd tego, co jest w niej dostępne. Zacznij stąd:

https://docs.python.org/3/library/index.html.

Jeśli to, czego potrzebujesz, znajduje się w *bibliotece standardowej*, nie trać czasu na ponowne pisanie tego od zera. Wykorzystuj (a także rozszerzaj) to, co jest już dostępne. Poza przeglądaniem dokumentacji Pythona zerknij też na popularną stronę *Module of the Week* (moduł tygodnia), na której *Doug Hellmann* zamieszcza swoje świetnie materiały dotyczące języka Python 3. Materiały te znajdziesz tutaj:

https://pymotw.com/3/.

Poniżej opisaliśmy kilka naszych ulubionych modułów *biblioteki standardowej*. Zauważ, że nie da się przecenić znaczenia wiedzy na temat tego, co znajduje się w *bibliotece standardowej*, jak również tego, co można osiągnąć za pomocą dostępnych w niej modułów.

Moduł collections

Moduł ten zapewnia dostępne do zaimportowania struktury danych wykraczające poza to, co oferują zwykłe listy, krotki, słowniki i zbiory. Wspomniany moduł oferuje wiele ciekawych możliwości. Oto skrócona lista elementów należących do modułu `collections`:

- `OrderedDict` — słownik, który zachowuje kolejność wstawiania danych;
- `Counter` — klasa, która bardzo ułatwia zliczanie różnych rzeczy;
- `ChainMap` — klasa, która łączy wiele różnych słowników w taki sposób, że zachowują się jak jeden słownik.

Moduł itertools

Wiesz już, że dostępna w języku Python pętla `for` jest świetna, a jeśli przerobimy ją na komprehencję, posługiwanie się pętlą staje się wręcz przezabawne. Moduł `itertools` zapewnia ogromną kolekcję narzędzi umożliwiających tworzenie własnych iteracji. Moduł ten oferuje wiele ciekawych możliwości, lecz koniecznie sprawdź metody `product`, `permutations` oraz `combinations` (a gdy to już zrobisz, usiądź wygodnie i podziękuj swoim szczęśliwym gwiazdom za to, że nie musiałeś pisać kodu którejkolwiek z tych metod samodzielnie).

Moduł functools

Biblioteka `functools` zapewnia kolekcję funkcji wyższego rzędu (czyli funkcji, które przyjmują obiekty funkcji jako swoje argumenty). Naszą ulubioną jest funkcja `partial`, która umożliwia „zamrożenie" wartości argumentów dla istniejącej funkcji, a następnie wywołuje tę funkcję z nową, określoną przez Ciebie nazwą. Nie będziesz wiedział, co tracisz, dopóki nie spróbujesz użyć tego mechanizmu.

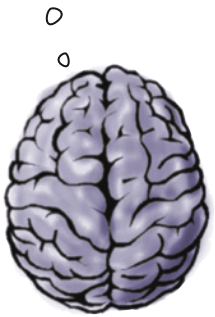
7. Równoległe wykonywanie Twojego kodu

W rozdziale 11 3/4. korzystałeś z wątku w celu rozwiązania problemu związanego z oczekiwaniem. Wątki nie są jedynym dostępnym rozwiązaniem, gdy chodzi o równoległe wykonywanie kodu w Twoich programach, chociaż — jeśli mamy być szczerzy — są one najczęściej wykorzystywane i nadużywane ze wszystkich dostępnych technik. W tej książce celowo opisaliśmy tylko najprostszy sposób zastosowania wątków.

Istnieją również inne dostępne możliwości, z których możesz skorzystać, gdy stwierdzisz, że Twój kod powinien robić więcej niż jedną rzecz naraz. Nie każdy program wymaga zastosowania tego rodzaju technik, dobrze jednak wiedzieć, że język Python oferuje w tym obszarze różne opcje do wyboru, jeśli tylko zajdzie taka potrzeba.

Zdajesz sobie sprawę, że jestem tu tylko jeden, prawda? A mimo to oczekujesz, że wykonam i zrozumiem wiele zadań obliczeniowych naraz?!?!?!

Oprócz modułu `threading` warto sprawdzić też inne, z których kilka prezentujemy poniżej (radzimy Ci też odwiedzić wspomniany w poprzednim podrozdziale (czyli na poprzedniej stronie) serwis *Douga Hellmanna*, który opublikował szereg świetnych artykułów na temat tych modułów).



- Moduł `multiprocessing` — pozwala Ci tworzyć wiele procesów Pythona, które — jeśli dysponujesz więcej niż jednym rdzeniem procesora — umożliwiają rozłożenie obciążenia obliczeniowego pomiędzy wiele jednostek CPU.
- Moduł `asyncio` — umożliwia Ci definiowanie współbieżności poprzez tworzenie i specyfikowanie współprogramów (koprocedur). To stosunkowo nowy dodatek do Pythona 3, dlatego — dla wielu programistów — stanowi bardzo świeżą ideę (i jest jeszcze zbyt wcześnie, aby go oceniać).
- Moduł `concurrent.futures` — pozwala zarządzać kolekcją zadań i uruchamiać je równolegle.

Które z tych modułów będą dla Ciebie odpowiednie, okaże się, gdy wypróbujesz każdy z nich z własnym kodem. Tylko tak przekonasz się o ich możliwościach i wadach.

Nowe słowa kluczowe: async i await

Słowa kluczowe `async` oraz `await` zostały dodane do języka w Pythonie 3.5. Zapewniają one standardowy sposób tworzenia współprogramów (koprocedur).

Słowa kluczowego `async` można użyć przed istniejącymi instrukcjami `for`, `with` oraz `def` (użycie go z tym ostatnim słowem kluczowym przyciąga jak do tej pory najwięcej uwagi). Słowo kluczowe `await` może być wykorzystywane przed (prawie) każdym innym kodem. W 2017 roku obydwa te słowa kluczowe nadal są stosunkowo nowe i programiści Pythona na całym świecie w dalszym ciągu poznają ich możliwości i uczą się z nich korzystać.

Dokumentacja języka Python została zaktualizowana o informacje dotyczące tych nowych słów kluczowych, ale — naszym zdaniem — najlepsze opisy ich zastosowania (oraz szaleństwa, jakie wprowadzają) znajdziesz, wyszukując w serwisie *YouTube* wszelkie materiały na ten temat zamieszczone przez *Davida Beazleya*. **Ostrzegamy:** wystąpienia Davida są zawsze doskonałe, ale zwykle poświęcone są bardziej zaawansowanym zagadnieniom w ekosystemie języka Python.

Porady maniaka

„GIL" to skrót od angielskich słów *Global Interpreter Lock*, które znaczą „globalna blokada interpretera". Jest to wewnętrzny mechanizm używany przez interpreter w celu zapewnienia stabilności. Jego ciągłe zastosowanie w interpreterze jest tematem wielu dyskusji i debat wewnątrz społeczności programistów Pythona.

Wykłady Davida poświęcone pythonowemu mechanizmowi *GIL* są przez wielu uważane za klasykę, świetne są również jego książki; więcej informacji na ich temat znajdziesz w dodatku E.

8. Graficzne interfejsy użytkownika z biblioteką tkinter (i zabawa z żółwiami)

Python zapewnia kompletną bibliotekę o nazwie `tkinter` (*Tk interface*) umożliwiającą budowanie graficznych interfejsów użytkownika (GUI) działających na różnych platformach. Być może nie zdajesz sobie z tego sprawy, ale od początku lektury tej książki korzystałeś z aplikacji, która powstała przy użyciu biblioteki `tkinter` — chodzi o IDLE.

Miłą rzeczą związaną z biblioteką `tkinter` jest to, że jest ona preinstalowana (i gotowa do użycia) wraz z każdą instalacją Pythona zawierającą środowiska IDLE (czyli niemal ze wszystkimi). Mimo to `tkinter` nie cieszy się tak powszechnym zastosowaniem (ani miłością), na jakie zasługuje, ponieważ wiele osób uważa, że biblioteka ta jest zbyt nieporęczna (w porównaniu z niektórymi alternatywnymi rozwiązaniami niezależnymi). Niemniej jednak — i jak pokazuje przykład IDLE — można za jej pomocą utworzyć przydatne i używalne programy. (Czy wspomnieliśmy już, że biblioteka `tkinter` jest preinstalowana i gotowa do użycia?).

Jednym z takich zastosowań jest moduł `turtle` (który również stanowi część *biblioteki standardowej*). Zacytujmy tu odpowiedni fragment dokumentacji Pythona: *Grafika żółwia jest popularnym sposobem prezentowania dzieciom podstaw programowania. Początkowo była częścią języka Logo opracowanego przez Wallyego Feurziga oraz Seymoura Paperta w 1966 roku*. Programiści (czyli głównie dzieci, ale również początkujący będą się świetnie bawić) mogą korzystać z poleceń takich jak `left`, `right`, `pendown` czy `penup` i mogą rysować na kanwie GUI (zapewnianej przez bibliotekę `tkinter`).

Poniżej znajdziesz kod niewielkiego programu, który jest nieznacznie zmienioną wersją przykładu znajdującego się w dokumentacji modułu `turtle`.

Ten niewielki program poza tym, że prezentuje moduł turtle w akcji, demonstruje również zastosowanie zapewnianych przez Pythona pętli while oraz instrukcji break. Obydwie te konstrukcje działają tak, jak możesz się tego spodziewać, jednak nie są używane nawet w przybliżeniu tak często, jak pętle for i komprehencje.

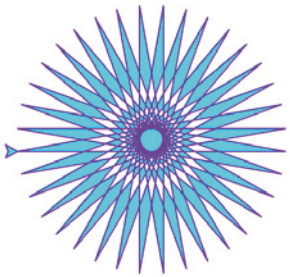
tksample.py - C:/Users/lsuma/De...

File Edit Format Run Options Window Help

```
from turtle import *

color('purple', 'cyan')
begin_fill()

while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break

end_fill()
done()
```

Ln: 15 Col: 0

A gdy ten niewielki program wykorzystujący moduł `turtle` zostanie uruchomiony, coś bardzo pięknego jest rysowane na ekranie.



Wiemy, że stać Cię na jeszcze więcej, więc dlaczego nie miałbyś wypróbować możliwości modułu turtle?

9. Kod nie jest skończony, dopóki nie jest przetestowany

W tej książce ledwo wspomnieliśmy o testach automatycznych; jedyna wzmianka dotyczyła sprawdzania zgodności kodu z wytycznymi zawartymi w dokumencie *PEP 8*, które przeprowadzaliśmy za pomocą narzędzia `py.test` (pod koniec rozdziału 4.). To nie dlatego, że uważamy testy automatyczne za nieistotne. **Naszym zdaniem testy automatyczne są bardzo ważne.** Temat ten jest tak istotny, że są mu poświęcone całe książki.

Mimo to celowo pominęliśmy opis narzędzi do testów automatycznych. Nie ma to nic wspólnego z naszym osobistym stosunkiem do testów automatycznych (naprawdę *są* one bardzo istotne). Jednak gdy uczysz się podstaw programowania w nowym języku, wprowadzenie tematyki testów automatycznych może uczynić więcej szkody niż pożytku i bardziej zagmatwać sprawy niż wyjaśnić, jako że tworzenie testów wymaga dobrego zrozumienia rzeczy, która ma być im poddana, a jeśli tak się składa, że „rzecz" ta jest nowym językiem programowania, którego się uczysz... cóż, chyba już widzisz, dokąd tu zmierzamy, prawda? To trochę jak z kurą i jajkiem. Co powinno być pierwsze: nauka tworzenia kodu czy też nauka testowania kodu, który uczysz się tworzyć?

Hej, nie patrz się na mnie... To nie ja umieściłam tutaj to zdanie.

Oczywiście teraz, gdy jesteś już prawdziwym programistą Pythona, możesz poświęcić trochę czasu, aby zrozumieć, jak *biblioteka standardowa* Pythona ułatwia Ci testowanie Twojego kodu. Powinieneś przyjrzeć się dwóm wymienionym poniżej modułom (i zastanowić się nad ich zastosowaniem).

- Moduł `doctest` — moduł ten umożliwia Ci osadzenie Twoich testów w tekstach dokumentujących Twoich modułów, co wcale nie jest aż tak dziwnym pomysłem, jak może się początkowo wydawać, i *jest* bardzo przydatne.
- Moduł `unittest` — bibliotekę o tej samej nazwie mogłeś już wykorzystywać w innym języku programowania; Python zapewnia własną wersję tej biblioteki (która działa dokładnie tak, jak możesz tego oczekiwać).

Moduł `doctest` jest uwielbiany przez osoby, które z niego korzystają. Moduł `unittest` działa tak, jak większość innych bibliotek noszących tę nazwę, dostępnych w innych językach programowania, a większość programistów Pythona narzeka, że nie jest on wystarczająco *pythoniczny*. Doprowadziło to do utworzenia wielce popularnego narzędzia `py.test` (o którym więcej napiszemy w kolejnym dodatku).

Nawet nie waż się przechodzić do następnej strony, dopóki nie przemyślisz, jak będziesz korzystał z oferowanych przez Pythona narzędzi do przeprowadzania testów automatycznych w celu sprawdzenia poprawności kodu, który tworzysz.

10. Debuguj, debuguj, debuguj

Wybaczymy Ci, jeśli myślisz, że ogromna większość programistów Pythona dodaje wywołania funkcji `print` do swojego kodu, gdy coś idzie źle. W gruncie rzeczy nie będziesz daleki od prawdy, ponieważ rzeczywiście jest to całkiem popularna technika debugowania.

Inną techniką jest eksperymentowanie w wierszu poleceń powłoki Pythona, które — jeśli się dobrze zastanowić — bardzo przypomina sesję debugowania *bez* związanej zwykle z debugowaniem konieczności męczącego przeglądania śladów i wstawiania punktów wstrzymania. Nie da się ocenić, jak wielki wpływ na wydajność programistów Pythona ma wiersz poleceń. Wiemy tylko, że jeśli w którymś z przyszłych wydań Pythona zostanie on usunięty, sprawy szybko zaczną się komplikować.

Wszystkiego o śladach i punktach wstrzymania dowiesz się, czytając dokumentację modułu pdb.

Jeśli masz do czynienia z kodem, który nie robi tego, co — jak sądzisz — powinien robić, a dodanie wywołań funkcji `print` i eksperymentowanie w wierszu poleceń powłoki Pythona nie czyni Cię ani trochę mądrzejszym, rozważ skorzystanie z dostępnego w Pythonie debuggera, czyli modułu `pdb`.

Debugger `pdb` można uruchamiać bezpośrednio z poziomu okna terminala Twojego systemu operacyjnego, korzystając z polecenia podobnego do przedstawionego poniżej (w którym *myprog.py* jest nazwą programu, który chcesz naprawić).

```
py -3 -m pdb myprog.py
```

Użytkownicy systemów operacyjnych innych niż Windows zamiast polecenia `py -3` jak zawsze muszą tu użyć polecenia `python3`.

Można również korzystać z modułu `pdb` z poziomu wiersza poleceń powłoki Pythona, co jest czymś najbliższym ucieleśnienia „najlepszych cech obydwu światów”, z czym naszym zdaniem dane Ci się będzie zetknąć. Szczegóły dotyczące sposobu działania tego rozwiązania, jak również informacje na temat wszystkich typowych poleceń debuggera (umożliwiających ustawienie punktu przerwania, pominięcie go, uruchomienie kodu itd.) znajdują się w dokumentacji, którą znajdziesz pod adresem:

https://docs.python.org/3/library/pdb.html.

Rozwiązanie `pdb` nie jest żadnym niechcianym dzieckiem ani też zbędnym dodatkiem; to wspaniały, wyposażony we wszystkie niezbędne funkcje debugger kodu Pythona (który jest w dodatku wbudowany w sam język).

Postaraj się, aby praktyczna wiedza na temat możliwości modułu pdb stała się elementem zestawu narzędzi, którymi się posługujesz.

Jeszcze więcej narzędzi, bibliotek i modułów

Wiemy, co sobie myślisz, czytając tytuł tego rozdziału.

Dlaczego, u licha, nie zatytułowali poprzedniego dodatku: *Dwadzieścia najważniejszych tematów, których nie poruszyliśmy*? Dlaczego *kolejne* 10? W poprzednim dodatku ograniczyliśmy się do opisu rzeczy, które są wbudowane w samego Pythona (jako część „baterii w zestawie" tego języka). W tym dodatku zarzucamy naszą sieć znacznie dalej, omawiając szereg technologii, które są dostępne, *ponieważ* Python istnieje. Mamy tu do dyspozycji mnóstwo interesujących rzeczy, a ich krótki przegląd — podobnie jak miało to miejsce w poprzednim dodatku — z pewnością nie zrobi Ci *najmniejszej* krzywdy.

1. Alternatywy dla wiersza poleceń powłoki Pythona

W całej tej książce z powodzeniem korzystaliśmy z wbudowanego wiersza poleceń powłoki Pythona (ze znakiem zachęty >>>), używając w tym celu okna systemowego terminala lub okna środowiska IDLE. Mamy nadzieję, że robiąc to, pokazaliśmy Ci, jak skuteczne może być stosowanie wiersza poleceń, gdy eksperymentujesz z różnymi koncepcjami, sprawdzasz możliwości bibliotek i wypróbowujesz kod.

Istnieje mnóstwo alternatyw dla wbudowanego wiersza poleceń powłoki Pythona, ale najwięcej uwagi przyciąga rozwiązanie o nazwie `ipython`. Jeśli poczujesz, że chciałbyś zrobić coś, czego nie da się wykonać w standardowym wierszu poleceń, sprawdź możliwości tego rozwiązania. Jest ono bardzo popularne i używa go wielu programistów Pythona, ale *największym* wzięciem cieszy się w społeczności naukowej.

Aby przekonać się, co można osiągnąć za pomocą programu `ipython` w porównaniu ze zwykłym wierszem poleceń powłoki Pythona, spójrz na przedstawiony poniżej krótki przykład zastosowania tego rozwiązania.

```
C:\Users\lsuma\Desktop\Python>ipython
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: nums = [1, 2, 3]

In [2]: squares = [n*n for n in nums]

In [3]: squares
Out[3]: [1, 4, 9]

In [4]:
```

Po zainstalowaniu programu ipython uruchom go w wierszu poleceń swojego systemu.

Łatwo jest tu rozpoznać dane wyjściowe generowane przez określone instrukcje (dzięki numeracji wiersza poleceń).

Poszczególne części kodu są wyświetlane za pomocą odpowiednich kolorów.

Więcej informacji na temat rozwiązania `ipython` znajdziesz na stronie *https://ipython.org*.

Istnieją również inne alternatywy dla standardowego wiersza poleceń powłoki Pythona, ale jedyną, której możliwości dorównują (naszym zdaniem) możliwościom programu `ipython`, jest `ptpython` (więcej informacji na ten temat można znaleźć tutaj: *https://pypi.org/project/ptpython/*). Jeśli lubisz korzystać z okna terminala tekstowego, ale szukasz czegoś bardziej „pełnoekranowego” niż `ipython`, przyjrzyj się rozwiązaniu `ptpython`. Nie zawiedziesz się na nim.

Tak jak w przypadku wszystkich modułów niezależnych, rozwiązania ipython oraz ptpython możesz zainstalować za pomocą narzędzia pip.

Pssst! Od czasu odkrycia programu ptpython Paul używa go bez przerwy.

2. Alternatywy dla środowiska IDLE

Nie boimy się przyznać, że mamy słabość do *IDLE*. Podoba nam się, że Python oferuje nie tylko całkiem użyteczny wiersz poleceń, lecz również zupełnie znośne, wieloplatformowe, wyposażone w graficzny interfejs użytkownika edytor i debugger. Tylko kilka innych popularnych języków programowania zapewnia podobne narzędzia jako część swojej standardowej instalacji.

Niestety środowisko *IDLE* poddawane jest ostrej krytyce przez społeczność Pythona, ponieważ prezentuje się dość biednie w porównaniu z bardziej rozbudowanymi, „profesjonalnymi" narzędziami. Naszym zdaniem to *niesprawiedliwe* porównanie, jako że *IDLE* nigdy nie miało konkurować z tego rodzaju rozwiązaniami. Głównym zadaniem *IDLE* jest umożliwienie nowym użytkownikom jak najszybszego rozpoczęcia pracy i radzi ono sobie z tym wręcz *doskonale*. Dlatego środowisko *IDLE* powinno cieszyć się większym szacunkiem w społeczności Pythona.

Oprócz *IDLE* masz do dyspozycji inne, bardziej profesjonalne IDE, jeśli tylko ich potrzebujesz. Do najbardziej popularnych wśród użytkowników Pythona należą:

- *Eclipse*: *https://www.eclipse.org*,
- *PyCharm*: *https://www.jetbrains.com/pycharm/*,
- *WingWare*: *https://wingware.com*.

Eclipse to całkowicie opensource'owe rozwiązanie, a więc nie będzie Cię kosztować więcej niż pobranie odpowiedniego instalatora. Jeśli jesteś już fanem tego IDE, wiedz, że doskonale radzi sobie ono z obsługą Pythona. Jeśli jednak jeszcze nie używasz środowiska *Eclipse*, nie radzimy Ci zaczynać tego robić, ponieważ istnieją rozwiązania *PyCharm* oraz *WingWare*.

Zarówno *PyCharm*, jak i *WingWare* są produktami komercyjnymi, których „wersje społecznościowe" można pobrać zupełnie za darmo (jednak z pewnymi ograniczeniami). W przeciwieństwie do *Eclipse*, które współpracuje z wieloma różnymi językami programowania, *PyCharm* i *WingWare* są przeznaczone specjalnie dla programistów Pythona i — jak wszystkie IDE — oferują doskonałe wsparcie dla pracy nad projektami, połączenia z narzędziami do zarządzania kodem źródłowym (takimi jak *git*), wsparcie dla zespołów, łącza do dokumentacji Pythona i tym podobne funkcje. Zachęcamy Cię do wypróbowania obydwu, a następnie wybrania tego, które będzie Ci bardziej odpowiadać.

Jeśli IDE nie jest rozwiązaniem dla Ciebie, nie martw się, ponieważ wszystkie ważniejsze edytory tekstu oferują doskonałe wsparcie dla programistów korzystających z języka Python.

Czego używa Paul?

Wybór Paula padł na edytor tekstowy `vim` (Paul używa programu *MacVim* na komputerach, na których programuje). Pracując nad projektami w Pythonie, Paul uzupełnia użycie edytora `vim` zastosowaniem narzędzia `ptpython` (gdy eksperymentuje z fragmentami kodu), jest też jednak fanem *IDLE*. Do lokalnej kontroli wersji Paul używa rozwiązania *git*.

Jeśli ma to jakieś znaczenie — Paul nie używa kompletnego IDE, ale jego studenci uwielbiają środowisko *PyCharm*. Paul korzysta również z programu *Jupyter Notebook* (i poleca go wszystkim), który został omówiony w następnym podrozdziale.

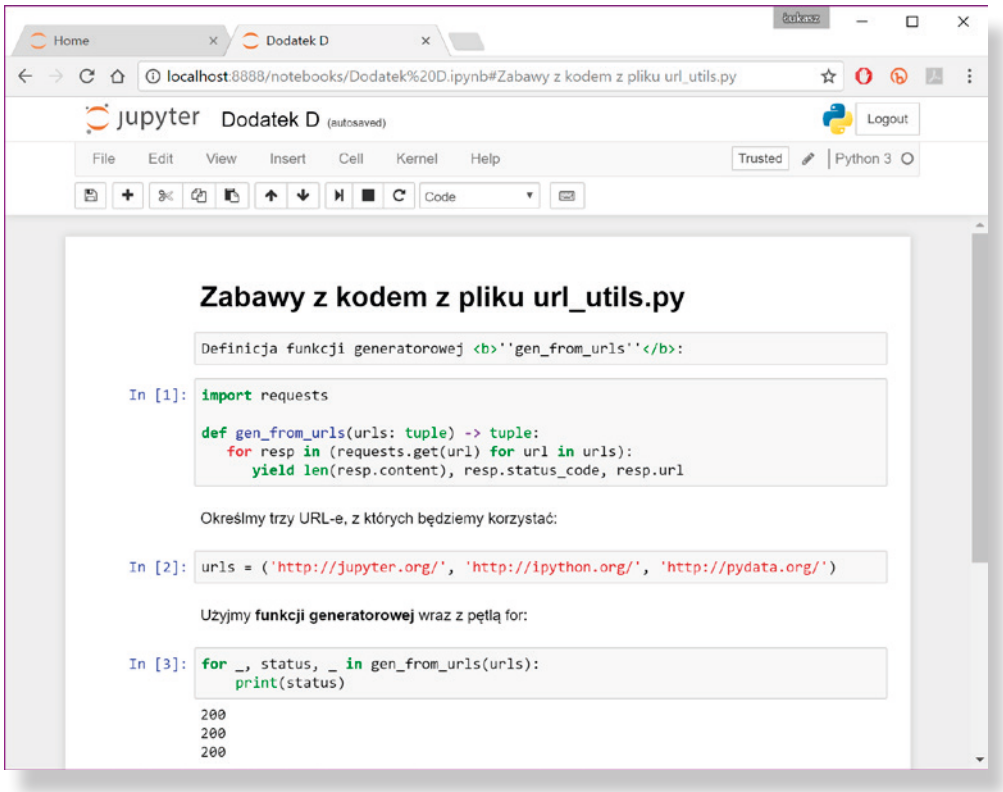
3. Jupyter Notebook: IDE dostępne z poziomu przeglądarki WWW

W podrozdziale 1. zwróciliśmy Twoją uwagę na rozwiązanie `ipython` (które jest doskonałą alternatywą dla standardowego wiersza poleceń powłoki Pythona). Ta sama ekipa, która je tworzy, odpowiada również za rozwój oprogramowania *Jupyter Notebook* (znanego wcześniej jako *iPython Notebook*).

Następna generacja rozwiązania Jupyter Notebook nosi nazwę Jupyter Lab i w momencie kończenia pracy nad tą książką była na etapie wersji alfa. Miej oko na projekt Jupyter Lab, ponieważ zapowiada się bardzo ciekawie.

Jupyter Notebook można opisać jako narzędzie oferujące możliwości rozwiązania `ipython` dostępne pod postacią interaktywnej strony WWW (noszącej ogólną nazwę „notebook", czyli „notatnik"). Niesamowitą cechą serwisu *Jupyter Notebook* jest to, że swój kod możesz edytować i uruchamiać wewnątrz notatnika i — jeśli odczuwasz taką potrzebę — możesz również dodawać do niego opisy i grafiki.

Poniżej znajdziesz trochę przedstawionego w rozdziale 12. kodu, który został uruchomiony wewnątrz serwisu *Jupyter Notebook*. Zwróć uwagę na to, że w notatniku dodaliśmy opis tekstowy, aby wskazać, co się dzieje.



Więcej informacji na temat oprogramowania *Jupyter Notebook* znajdziesz na poświęconej mu stronie (*http://jupyter.org*). Aby zainstalować to rozwiązanie na swoim komputerze, skorzystaj z narzędzia `pip`. Będziesz dzięki temu mógł poznać możliwości programu *Jupyter Notebook* i nie pożałujesz, że zdecydowałeś się na jego instalację, ponieważ to naprawdę *kapitalna* aplikacja napisana w Pythonie.

4. Przeprowadzanie analizy danych

Python przyjmuje się świetnie i jest wykorzystywany szczególnie mocno w **analizie danych** (ang. *data science*). W dziedzinie tej notuje wręcz lawinowy wzrost zastosowania.

Nie dzieje się tak przypadkiem. Narzędzia oferowane analitykom danych używającym języka Python są światowej klasy (stanowią też obiekt zazdrości wielu innych społeczności programistycznych). Świetną rzeczą dla naukowców niezajmujących się danymi jest to, że narzędzia lubiane przez specjalistów od danych znajdują wiele zastosowań poza obszarem *Big Data*.

Całe książki zostały napisane (i nadal są pisane) na temat stosowania Pythona w obszarze *analizy danych*. Choć radę tę możesz uznać za stronniczą, wydawnictwo *Helion* oferuje szereg interesujących książek na ten temat. *Helion* rozwinął swój biznes dzięki wyczuciu kierunku, w którym podąża branża informatyczna, oraz zapewnieniu dużej dawki doskonałych materiałów do nauki dostępnych dla tych, którzy chcą dowiedzieć się więcej.

Uwaga z działu marketingu: To oznacza, że dostali naszą notkę :)

Poniżej znajdziesz wybór kilku dostępnych bibliotek i modułów, które mogą Ci się przydać do analizowania danych (a w gruncie rzeczy również do przeprowadzania innych obliczeń naukowych). Jeśli nie zajmujesz się *analizą danych*, też przejrzyj te rzeczy, ponieważ oferują one sporo interesujących możliwości.

- Biblioteka `bokeh` — jest to zestaw narzędzi do publikowania interaktywnych grafik na stronach WWW.
- Biblioteka `matplotlib/seaborn` — to wszechstronny zestaw modułów graficznych (które można zintegrować z modułem `ipython` oraz rozwiązaniem *Jupyter Notebook*).
- Biblioteka `numpy` — oprócz innych interesujących funkcji zapewnia możliwość efektywnego przechowywania wielowymiarowych danych i manipulowania nimi. Jeśli jesteś fanem macierzy, pokochasz `numpy`.
- Biblioteka `scipy` — jest to zestaw modułów naukowych zoptymalizowanych pod kątem analiz danych liczbowych. Biblioteka ta uzupełnia i rozbudowuje możliwości oferowane przez rozwiązanie `numpy`.
- Biblioteka `pandas` — jeśli przesiadasz się na Pythona z języka *R*, biblioteka `pandas` sprawi, że poczujesz się jak w domu. Biblioteka ta zapewnia zoptymalizowane analityczne struktury danych oraz odpowiednie narzędzia (a sama została zbudowana na bazie bibliotek `numpy` i `matplotlib`). Potrzeba użycia biblioteki `pandas` stanowi powód, dla którego osoby zajmujące się analizą danych przyłączają się do społeczności programistów Pythona (i prawdopodobnie będzie tak jeszcze bardzo, bardzo długo). Biblioteka `pandas` to kolejna *kapitalna* aplikacja Pythona.
- Biblioteka `scikit-learn` — zawiera zestaw zaimplementowanych w języku Python algorytmów i rozwiązań związanych z uczeniem maszynowym.

Sacrébleu! Ośmielają się pytać, skąd wiem, że mój przepis na zupę jest najlepszy?!? Dlaczego... Przeprowadziłem szybką analizę danych za pomocą biblioteki pandas, po czym opublikowałem jej wyniki za pomocą narzędzia Jupyter Notebook. Voilà — teraz wszyscy już wiedzą.

Większość wymienionych tu bibliotek i modułów instaluje się za pomocą narzędzia `pip`.

Najlepszym miejscem do rozpoczęcia poznawania możliwości języka Python w zakresie *analizy danych* jest serwis *PyData*, znajdujący się pod adresem *https://pydata.org*. Kliknij widoczne na tej stronie łącze *Downloads* (pliki do pobrania), a następnie zachwyć się tym, co jest tam dostępne (wszystkie dostępne tam pliki mają charakter open source). Baw się dobrze!

5. Rozwiązania do tworzenia aplikacji WWW

Język Python ma bardzo silną pozycję w obszarze aplikacji internetowych, ale framework *Flask* (wraz z silnikiem *Jinja2*) nie stanowi jedynego rozwiązania, za pomocą którego można je tworzyć (mimo że *Flask* jest bardzo popularną opcją, zwłaszcza gdy potrzeby są umiarkowane).

Najbardziej znanym rozwiązaniem do budowania aplikacji WWW jest *Django*. Nie korzystaliśmy z niego w tej książce, ponieważ (w przeciwieństwie do *Flaska*) zanim będzie można opracować pierwszą aplikację wykorzystującą *Django*, trzeba poznać i zrozumieć ten framework dość dobrze (co za tym idzie, do tego rodzaju książki, której celem jest *solidne* przedstawienie podstaw Pythona, *Django* nadaje się dość słabo). Jednakże *Django* nie bez powodu jest bardzo popularnym frameworkiem wśród programistów Pythona. Przyczyną tego stanu rzeczy jest to, że jest naprawdę doskonałe.

Jeśli uważasz się za programistę aplikacji WWW, powinieneś poświęcić czas (przynajmniej) na przeczytanie instrukcji *Django*. Dzięki temu zdobędziesz więcej wiedzy, która pozwoli Ci świadomie zdecydować, czy powinieneś trzymać się frameworka *Flask*, czy też przejść na *Django*.

Jeśli wybierzesz *Django*, dołączysz do bardzo dobrego towarzystwa: społeczność użytkowników *Django* stanowi tak wielką grupę w szerszym gronie użytkowników Pythona, że jest w stanie organizować własną konferencję, która nosi nazwę *DjangoCon* i odbywa się w Europie, Stanach Zjednoczonych oraz Australii. Więcej informacji na ten temat znajdziesz pod poniższymi adresami:

- strona główna *Django* (na której znajdziesz łącze do instrukcji):
  https://www.djangoproject.com,
- DjangoCon Europe:
  https://djangocon.eu,
- DjangoCon US:
  https://djangocon.us,
- DjangoCon Australia:
  http://djangocon.com.au.

Czekaj, to jeszcze nie wszystko

Oprócz *Flaska* i *Django* istnieją jeszcze inne frameworki sieciowe (wiemy, że z pewnością pominiemy tu jakiś, który okaże się czyimś ulubionym rozwiązaniem). Tymi, o których słyszymy najczęściej, są: *Pyramid*, *TurboGears*, *web2py*, *CherryPy* oraz *Bottle*. Pełną listę frameworków sieciowych dostępnych dla użytkowników Pythona znajdziesz w serwisie wiki poświęconym temu językowi:

https://wiki.python.org/moin/WebFrameworks.

6. Korzystanie z danych sieciowych

W rozdziale 12. korzystaliśmy przez chwilę z biblioteki `requests`, aby zademonstrować, jak fajny był nasz generator (w porównaniu z odpowiadającą mu komprehencją). Nie przypadkiem postanowiliśmy użyć właśnie tej biblioteki. Jeśli zapytasz programistów Pythona zajmujących się opracowywaniem aplikacji sieciowych o ich ulubiony moduł PyPI, większość z nich odpowie jednym słowem: „requests”.

Moduł `requests` umożliwia korzystanie z serwisów sieciowych i HTTP za pośrednictwem prostego, lecz potężnego API Pythona. Nawet jeśli w swojej codziennej pracy nie używasz bezpośrednio sieci, możesz się dużo nauczyć dzięki samemu tylko czytaniu kodu tego modułu (cały projekt `requests` jest uważany za swojego rodzaju klasę mistrzowską pod względem tego, jak powinno się pisać kod w sposób właściwy dla języka Python).

PyPI, czyli indeks pakietów Pythona, jest dostępny pod adresem https://pypi.org/.

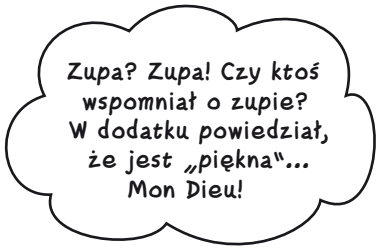
Więcej informacji na temat biblioteki `requests` znajdziesz pod adresem:

http://docs.python-requests.org/en/master/.

Zeskrob te dane sieciowe!

Jako że WWW jest przede wszystkim platformą tekstową, Python zawsze świetnie radził sobie w tym obszarze, a *biblioteka standardowa* zapewnia moduły umożliwiające korzystanie z JSON-a, HTML-a, XML-a i innych, podobnych do nich formatów tekstowych, jak również ze wszystkich liczących się protokołów internetowych. Przejrzyj przedstawione poniżej działy dokumentacji Pythona w celu zapoznania się z listą oferowanych w ramach *biblioteki standardowej* modułów, które powinny najbardziej zainteresować programistów aplikacji sieciowych:

- obsługa danych internetowych:
  https://docs.python.org/3/library/netdata.html,
- narzędzia do przetwarzania strukturalnego kodu znaczników:
  https://docs.python.org/3/library/markup.html,
- protokoły internetowe i ich obsługa:
  https://docs.python.org/3/library/internet.html.



Jeśli okaże się, że musisz korzystać z danych, które są dostępne jedynie na statycznych stronach internetowych, prawdopodobnie będziesz chciał *zeskrobać* (ang. *scrape*) te dane (więcej informacji o tej technice znajdziesz tu: *http://www.kesser.hekko.pl/web-scraping-w-teorii-czyli-nie-samym-kodem-czlowiek-zyje/*, a po angielsku tutaj: *https://en.wikipedia.org/wiki/Web_scraping*). Dla programistów Pythona dostępne są dwa niezależne moduły, których zastosowanie może zaoszczędzić Ci mnóstwo czasu:

- *Beautiful Soup*:
  https://www.crummy.com/software/BeautifulSoup/,
- *Scrapy*:
  http://scrapy.org.

Wypróbuj obydwa moduły, przekonaj się, który z nich najlepiej rozwiązuje Twój problem, a następnie weź się do roboty.

7. Więcej źródeł danych

Aby nasze przykłady były jak najbardziej rzeczywiste (będąc jednak również odpowiednio prostymi), używaliśmy w tej książce bazy danych *MySQL*. Jeśli spędzasz mnóstwo czasu, korzystając z języka SQL (niezależnie od tego, jaki jest Twój ulubiony producent bazy danych), przerwij to, co w tej chwili robisz, i poświęć dwie minuty na zainstalowanie za pomocą narzędzia `pip` modułu `sqlalchemy`; może się okazać, że to najlepiej zainwestowane w instalację dwie minuty w *całym Twoim życiu*.

Moduł `sqlalchemy` ma dla maniaków SQL-a takie znaczenie, jakie biblioteka `requests` ma dla maniaków rozwiązań sieciowych: jest po prostu niezbędny. Projekt *SQL Alchemy* zapewnia zestaw rozwiązań wysokiego poziomu zainspirowanych językiem Python, umożliwiających korzystanie z danych tabelarycznych (takich, jakie przechowywane są w bazach danych *MySQL*, *PostgreSQL*, *Oracle* czy *SQL Server*). Jeśli podobało Ci się to, co robiliśmy z modułem `DBcm`, wprost pokochasz *SQL Alchemy*, który uważa się za *właściwy* zestaw narzędzi bazodanowych dla użytkowników języka Python.

Więcej informacji na temat tego projektu znajdziesz na stronie:

http://www.sqlalchemy.org.

Inne niż SQL technologie bazodanowe

Nie wszystkie dane, których będziesz kiedyś potrzebował, będą zapisane w SQL-owej bazie danych, dlatego z pewnością zdarzą się czasem sytuacje, gdy rozwiązanie *SQL Alchemy* w niczym Ci nie pomoże. NieSQL-owe bazy danych uznaje się obecnie za pełnoprawne składniki wszystkich centrów danych, czego klasycznym i najczęściej używanym przykładem może być baza *MongoDB* (choć rozwiązań tego rodzaju jest bardzo wiele).

Jeśli okaże się, że korzystasz z danych, które otrzymujesz w formacie JSON lub innym formacie nietabelarycznym (ale mających jakąś strukturę), baza *MongoDB* (lub inna podobna) może być tym, czego szukasz. Więcej informacji na jej temat znajdziesz pod adresem:

https://www.mongodb.com.

Przekonaj się też, co w zakresie programistycznej obsługi bazy *MongoDB* oferuje język Python wraz ze sterownikiem bazy danych `pymongo`, który znajdziesz na stronie dokumentacji *PyMongo* dostępnej pod adresem:

https://api.mongodb.com/python/current/.

8. Narzędzia programistyczne

Niezależnie od tego, jak dobrze oceniasz swój kod, na pewno mogą w nim występować bugi.

W sytuacji gdy tak jest, Python oferuje szereg możliwości, które powinny Ci pomóc. Są wśród nich: wiersz poleceń powłoki, debugger `pdb`, środowisko *IDLE*, instrukcje `print`, moduły `unittest` oraz `doctest`. Jeśli one Ci nie wystarczają, dostępne są też moduły niezależne, które mogą pomóc.

Czasami zdarzy się, że popełnisz jakiś klasyczny błąd, który wcześniej przytrafił się już wszystkim innym. Może być też tak, że zapomnisz o zaimportowaniu jakiegoś niezbędnego modułu, a problem nie ujawni się aż do czasu, gdy zaprezentujesz swój wspaniały kod w sali pełnej obcych osób (ups).

Aby pomóc Ci uniknąć tego rodzaju wpadek, pobierz narzędzie o nazwie *PyLint* do analizowania kodu Pythona, które znajdziesz pod adresem:

https://www.pylint.org.

Narzędzie *PyLint* umożliwia przeanalizowanie kodu i określenie, co może być z nim nie tak, jeszcze zanim uruchomisz ten kod po raz pierwszy.

Jeśli skorzystasz z możliwości oferowanych przez to rozwiązanie, zanim uruchomisz swój kod przed dużym gronem nieznajomych, być może uda Ci się oszczędzić sobie trochę wstydu. *PyLint* może również nieco urazić Twoje uczucia, ponieważ nikt nie lubi dowiadywać się, że jego kod nie jest wystarczająco dobry. Warto jednak spokojnie znieść tego rodzaju cios (choć może powinniśmy napisać po prostu, że *cios ten jest lepszy niż publiczny wstyd*).

Więcej wsparcia również dla testów

W dodatku C omówiliśmy wbudowaną w Pythona obsługę mechanizmu testów automatycznych. Istnieją również inne narzędzia tego rodzaju i wiesz już, że należy do nich rozwiązanie `py.test` (wiesz już, ponieważ korzystaliśmy z niego we wcześniejszej części tej książki w celu sprawdzenia naszego kodu pod kątem zgodności z wytycznymi zebranymi w dokumencie *PEP 8*).

Frameworki do testów pod pewnym względem przypominają trochę frameworki sieciowe: każdy ma swój ulubiony. Jednak większość programistów Pythona preferuje `py.test`, dlatego zachęcamy Cię do przyjrzenia mu się bliżej:

https://docs.pytest.org/en/latest/.

9. Kivy: nasz wybór w kategorii „Najfajniejszy projekt wszech czasów"

Obszarem, w którym język Python nie jest tak mocny, jak mógłby być, jest świat dotykowych urządzeń mobilnych. Istnieje mnóstwo powodów tego stanu rzeczy (nie będziemy się tu w nie wgłębiać). Wystarczy stwierdzić, że w momencie publikacji tej książki utworzenie za pomocą samego tylko Pythona aplikacji działającej pod kontrolą systemu operacyjnego *Android* lub *iOS* wciąż jest nie lada wyczynem.

Istnieje jeden projekt, którego twórcy próbują na tej niwie osiągnąć pewien postęp, i jest nim *Kivy*.

Kivy to biblioteka Pythona, która umożliwia pisanie aplikacji wykorzystujących interfejsy wielodotykowe. Wejdź na główną stronę projektu, aby przekonać się, co on oferuje:

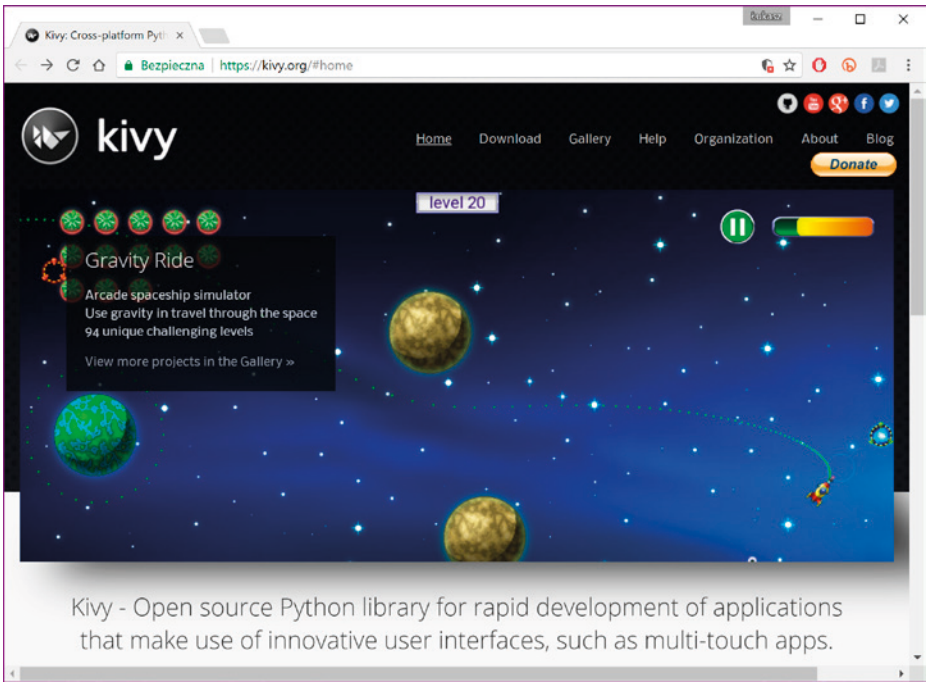
https://kivy.org.

Gdy się tam znajdziesz, kliknij łącze *Gallery* (galeria) i usiądź na chwilę, gdy strona będzie się ładowała. Jeśli jakiś projekt przyciągnie Twoją uwagę, kliknij odpowiednią grafikę, aby uzyskać więcej informacji na jego temat i zobaczyć demo. Oglądając demo, pamiętaj o następującej rzeczy: *wszystko, na co patrzysz, zostało napisane wyłącznie w języku Python*. Łącze *Blog* również prowadzi do interesujących materiałów.

Naprawdę świetną sprawą jest to, że gdy korzystając z *Kivy*, napiszesz już raz kod swojego interfejsu użytkownika, będziesz mógł *bez zmian* wdrożyć go na każdej platformie obsługiwanej przez to rozwiązanie.

Jeśli szukasz projektu Pythona, w który chciałbyś się zaangażować, rozważ poświęcenie swojego czasu na pracę nad *Kivy*: to wspaniała i wymagająca technicznie inicjatywa, którą rozwija doskonały zespół. Niezależnie od wszystkiego na pewno nie będziesz się tu nudził.

Zrzut okna przeglądarki wyświetlającej stronę główną projektu Kivy w stanie, w jakim była ona w połowie 2017 roku, przedstawia jedno z wdrożeń rozwiązania: rozbudowaną grę, stanowiącą symulator lotu statku kosmicznego w polu grawitacyjnym.



10. Alternatywne implementacje

Wiesz już z podrozdziału 1., wchodzącego w skład dodatku C, że istnieje więcej niż jedno wydanie języka Python (są to: Python 2 *oraz* Python 3). Oznacza to, że dostępne są *co najmniej* dwa interpretery Pythona: jeden, który umożliwia uruchamianie kodu napisanego w Pythonie 2, oraz drugi, pozwalający uruchamiać kod opracowany w Pythonie 3 (czyli ten, z którego korzystaliśmy w całej tej książce). Gdy pobierzesz ze strony Pythona i zainstalujesz jeden z tych interpreterów (jak robiłeś to w trakcie lektury dodatku A), będziesz miał do czynienia z interpreterem określanym jako implementacja referencyjna CPythona (ang. *CPython reference implementation*). *CPython* to wersja Pythona dystrybuowana przez *programistów rdzenia Pythona*, a jej nazwa wzięła się stąd, że jest ona napisana w przenośnym kodzie C, dzięki czemu może być łatwo przenoszona na inne platformy komputerowe. Jak przekonałeś się, czytając dodatek A, możesz pobrać instalatory dla systemów operacyjnych *Windows* oraz *Mac OS X*, jak również znaleźć preinstalowany interpreter wewnątrz swojej ulubionej dystrybucji systemu *Linux*. Wszystkie te interpretery powstały na bazie *CPythona*.

Python ma charakter open source, więc każdy może swobodnie pobrać kod *CPythona* i zmienić go w dowolny sposób. Programiści mogą także zapoznać się ze specyfikacją Pythona i napisać własny interpreter kodu tego języka, korzystając w tym celu z dowolnego języka programowania, który wybiorą, z zastosowaniem dowolnych technik kompilacji, jakie tylko przyjdą im do głowy, a także z wykorzystaniem dowolnej platformy. Choć nie jest to zadanie dla osób o słabych nerwach, wielu programistów się go podejmuje (a niektórzy określają je nawet jako „zabawę”). Oto krótkie opisy oraz adresy stron internetowych kilku bardziej aktywnych projektów tego rodzaju:

- *PyPy* (wymawiane jako „paj-paj”) to eksperymentalny kompilator (a właściwie poligon doświadczalny) Pythona 2 i Pythona 3. *PyPy* przyjmuje kod i uruchamia go dzięki przeprowadzanej dokładnie na czas kompilacji, w wyniku której powstaje produkt końcowy, działający w wielu sytuacjach szybciej niż w przypadku zastosowania *CPythona*. Więcej informacji na temat tego rozwiązania znajdziesz pod adresem:

  http://pypy.org.

- *IronPython* to wersja Pythona 2 przeznaczona na platformę *.NET*:

  http://ironpython.net.

- *Jython* to wersja Pythona 2, która działa w środowisku *JVM Javy*:

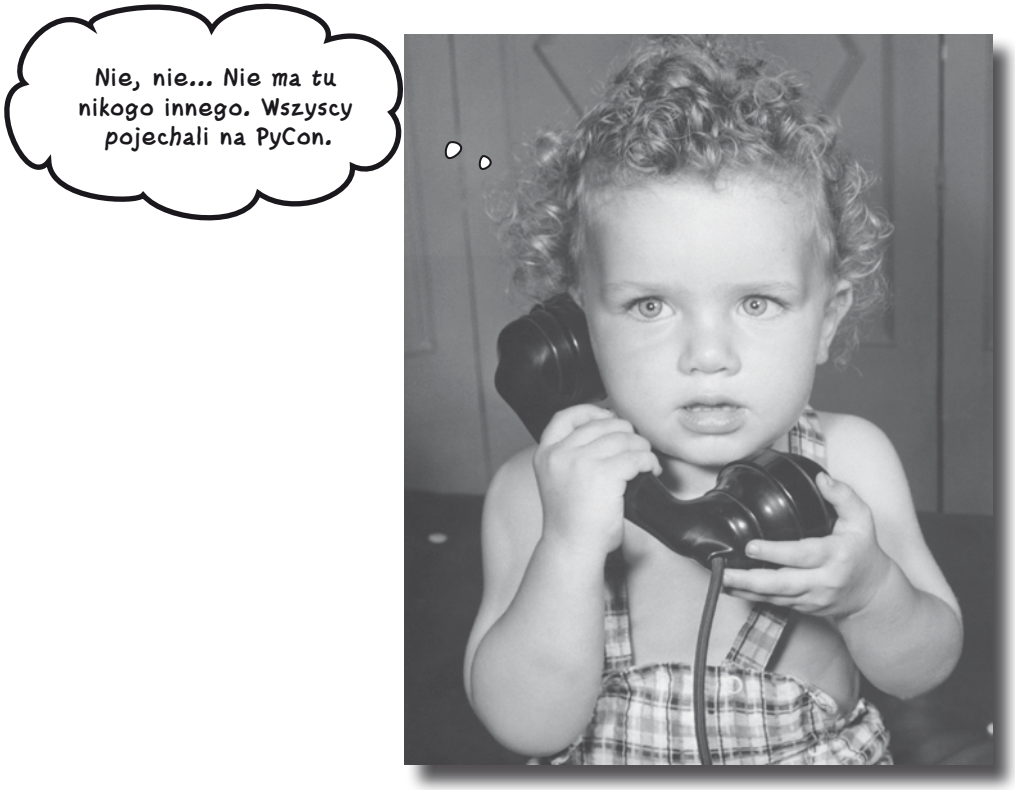
  http://www.jython.org.

- *MicroPython* to wersja Pythona 3 przeznaczona do użytku na mikrokontrolerze *pyboard*, który jest nie większy niż Twoje dwa złączone razem kciuki i być może jest jedną z najfajniejszych małych rzeczy, jakie kiedykolwiek zdarzyło Ci się widzieć. Zajrzyj na stronę tego projektu:

  http://micropython.org.

Pomimo istnienia tych wszystkich alternatywnych interpreterów Pythona większość programistów używających tego języka jest naprawdę zadowolona z *CPythona*. Coraz więcej osób wybiera też Pythona 3.

Społeczność Pythona



Python to coś znacznie więcej niż tylko doskonały język programowania.

To również wspaniała społeczność. Społeczność Pythona jest serdeczna, różnorodna, otwarta, przyjazna, hojna i zawsze chętnie dzieli się swoją wiedzą. Jesteśmy w szoku, że nikt nie pomyślał do tej pory, aby umieścić to wszystko na jakiejś laurce! A tak poważnie, programowanie w Pythonie nie sprowadza się wyłącznie do samego języka. Wokół niego wyrósł cały ekosystem, na który składają się doskonałe książki, blogi, serwisy WWW, konferencje, spotkania, grupy użytkowników i prawdziwe osobowości. W tym dodatku dokonamy przeglądu społeczności Pythona i wskażemy, co może Ci ona zaoferować. Nie siedź samotnie i nie programuj tylko dla siebie: **zaangażuj się**!

BDFL: Benevolent Dictator for Life

Guido van Rossum jest holenderskim programistą, którego darem dla świata jest język programowania Python (nad którym prace rozpoczął hobbystycznie pod sam koniec lat 80. ubiegłego wieku). Kontynuowany od tamtej pory rozwój, a także kierunek, w którym podąża ten język, określane są przez *programistów rdzenia Pythona*, wśród których Guido jest po prostu jednym z wielu (choć trzeba przyznać, że bardzo ważnym). Nadany Guido tytuł *Benevolent Dictator for Life* (czyli dobrotliwego, dożywotniego dyktatora) stanowi dowód uznania dla roli, jaką pełni on w codziennym życiu Pythona. Jeśli zobaczysz gdzieś litery BDFL w związku z językiem Python, to wiedz, że odnoszą się one do Guido.

Guido stwierdził, że nazwa „Python" stanowi ukłon (i porozumiewawcze mrugnięcie) w kierunku brytyjskiej grupy telewizyjnych komików, znanej jako *Latający Cyrk Monty Pythona*, co pomaga wyjaśnić zastosowanie nazwy `spam` dla wielu zmiennych, które pojawiają się w dokumentacji Pythona.

Mimo wiodącej roli Guido **nie** jest on właścicielem Pythona; nikt nim nie jest. Jednak interesy języka są chronione przez PSF.

PSF: Python Software Foundation

PSF to organizacja non profit, która reprezentuje interesy Pythona i jest prowadzona przez nominowany i wybierany zarząd. PSF promuje i sponsoruje ciągły rozwój języka. Oto cytat z myśli przewodniej tej organizacji:

> *Misją Python Software Foundation jest promocja, ochrona i rozwój języka programowania Python oraz wsparcie i ułatwianie wzrostu różnorodnej i międzynarodowej społeczności programistów Pythona.*

Miej swój głos — przyłącz się do PSF-u.

Każdy może zostać członkiem PSF-u i zaangażować się w pracę tej organizacji. Więcej informacji na ten temat znajdziesz w serwisie internetowym PSF-u:

https://www.python.org/psf/.

Jednym z głównych działań PSF-u jest zaangażowanie się w coroczną konferencję Pythona noszącą nazwę *PyCon* (oraz sygnowanie tejże konferencji).

PyCon: Konferencja Pythona

Każdy może wziąć udział (i przemawiać) na PyConie. W 2016 i 2017 roku konferencja odbyła się w Portland w stanie Oregon, gromadząc za każdym razem tysiące programistów Pythona (poprzednie dwie edycje imprezy miały miejsce w Montrealu w Kanadzie). PyCon to największa konferencja poświęcona Pythonowi, lecz nie jedyna. Podobne imprezy odbywają się na całym świecie; należą do nich zarówno niewielkie, regionalne spotkania (w których udział bierze kilkadziesiąt osób), jaki i średnie konferencje krajowe (w których uczestniczy kilkuset ludzi), a także wydarzenia, takie jak *EuroPython*, obejmujące swoim zasięgiem całe kontynenty (goszczące tysiące uczestników).

Zaangażuj się — weź udział w PyConie.

Aby się przekonać, czy w Twojej okolicy organizowany jest PyCon, wyszukaj w internecie słowo „PyCon" wraz z nazwą najbliższego miasta (lub kraju, w którym mieszkasz). Możesz zostać miło zaskoczony tym, co znajdziesz[1]. Udział w lokalnym PyConie to doskonała okazja do spotkania i rozmowy z programistami myślącymi podobnie jak Ty. Wiele wystąpień i sesji na różnych PyConach jest rejestrowanych, warto więc odwiedzić serwis *YouTube* i wpisać w jego wyszukiwarce słowo „PyCon", aby przekonać się, co można obejrzeć.

[1] W Polsce od wielu lat odbywa się konferencja PyCon PL. Impreza ta, która w 2017 roku już po raz dziesiąty jest organizowana przez społeczność skupioną wokół języka Python i wspierana przez Polską Grupę Użytkowników Linuksa, przyciąga co roku kilkuset uczestników z całego kraju. Więcej informacji na temat spotkania można znaleźć na stronie *https://pl.pycon.org — przyp. tłum.*

Tolerancyjna społeczność: szacunek dla różnorodności

Spośród wszystkich odbywających się obecnie konferencji programistycznych PyCon był jedną z pierwszych, które wprowadziły *Kodeks postępowania* i wywierały presję na jego przestrzeganie. *Kodeks postępowania* z 2017 roku możesz znaleźć na stronie:

https://us.pycon.org/2017/about/code-of-conduct/.

Stosowanie tego rodzaju reguł jest *bardzo dobrą rzeczą*. Coraz więcej niewielkich, regionalnych PyConów również przyjmuje *Kodeks postępowania*, co również jest bardzo mile widziane. Społeczność może się rozrastać, zachowując jednocześnie swoją siłę i otwartość, gdy istnieją przejrzyste zasady dotyczące tego, co jest dopuszczalne, a co takie nie jest. *Kodeks postępowania* pomaga zapewnić, że wszystkie odbywające się na świecie PyCony są tak gościnne, jak tylko się da.

Poza dokładaniem wszelkich starań w celu zapewnienia, żeby każdy uczestnik imprezy czuł się mile widziany, organizatorzy PyConu wystąpili z wieloma inicjatywami mającymi zwiększyć udział pewnych grup istniejących w społeczności Pythona, zwłaszcza tam, gdzie — tradycyjnie — są one niedostatecznie reprezentowane. Najbardziej znaną z tych inicjatyw jest *PyLadies*, którą powołano do życia w celu realizacji misji pomagania „większej liczbie kobiet w staniu się aktywnymi uczestniczkami i liderkami opensource'owej społeczności Pythona".

Jeśli masz szczęście, w Twojej okolicy funkcjonuje „sekcja" *PyLadies*; sprawdź to, zaczynając swoje poszukiwania od serwisu internetowego tej inicjatywy:

http://www.pyladies.com.

Tak jak sama społeczność Pythona, *PyLadies* zaczęła się jako niewielka inicjatywa, ale bardzo szybko rozrosła się tak, że obejmuje swoim zasięgiem cały świat (co jest naprawdę inspirujące)[2].

Pobudzaj i wspieraj różnorodność w społeczności Pythona.

Przyjdź ze względu na język, zostań ze względu na społeczność

Wielu nowych programistów Pythona wypowiada się na temat tego, jak otwarta i inkluzywna jest społeczność Pythona. Dużą część tego nastawienia zawdzięczamy przykładowi i przewodnictwu Guido, jego stanowczości, lecz również życzliwości. Istnieją także inni przywódcy społeczności oraz mnóstwo inspirujących historii.

Jedną z najbardziej inspirujących było wystąpienie *Naomi Ceder* na konferencji *EuroPython* (powtórzone następnie na innych konferencjach regionalnych, w tym na *PyCon Ireland*). Poniżej znajdziesz łącze do wykładu Naomi, do którego obejrzenia gorąco Cię namawiamy:

https://www.youtube.com/watch?v=cCCiA-IlVco.

W wystąpieniu tym Naomi dzieli się swoimi obserwacjami na temat życia z Pythonem i omawia wsparcie, jakie społeczność programistów używających tego języka daje różnorodności. Mówi także o tym, że wszyscy nadal mają przed sobą bardzo wiele pracy do wykonania.

Sposobem na bliższe poznanie społeczności może też być słuchanie niektórych podkastów tworzonych przez jej członków. Dwa takie podkasty opiszemy w następnym podrozdziale.

---

[2] Inicjatywa *PyLadies* obejmuje również Polskę. Informacje na temat jej działań znajdziesz na stronie internetowej *http://pyladies.pl — przyp. tłum.*

Podkasty dotyczące Pythona

Obecnie istnieją podkasty na *każdy* temat. W społeczności Pythona dostępne są dwa, które naszym zdaniem warte są subskrybowania i tego, by ich słuchać. Niezależnie, czy będziesz to robił w czasie prowadzenia samochodu, jazdy na rowerze, biegania czy relaksowania się, te dwa źródła zasługują na Twoją uwagę:

- *Talk Python to Me — https://talkpython.fm*,
- *Podcast.__init__ — http://pythonpodcast.com*.

Obserwuj obydwa te podkasty na Twitterze, powiedz o nim swoim znajomym i daj pełne wsparcie producentom tych podkastów. Zarówno *Talk Python to Me*, jak i *Podcast.__init__* są tworzone przez zwykłych członków społeczności Pythona dla dobra nas wszystkich (a *nie* dla zysku).

Biuletyny dotyczące Pythona

Jeśli podkasty do Ciebie nie trafiają, lecz nadal chcesz wiedzieć, co się dzieje w świecie Pythona — to wiedz, że dostępne są cotygodniowe newslettery, które mogą Ci pomóc być z nowościami na bieżąco:

- *Pycoder's Weekly — http://pycoders.com*,
- *Python Weekly — http://www.pythonweekly.com*,
- *Import Python — http://importpython.com/newsletter*.

Te wybrane przez nas biuletyny udostępniają łącza do wszelkiego rodzaju materiałów: blogów, vlogów, artykułów, książek, filmów wideo, wystąpień, nowych modułów i projektów. Cotygodniowe powiadomienia o nowościach trafiają prosto do Twojej skrzynki e-mailowej. Nie wahaj się więc i zapisz się na nie.

Oprócz fundacji, licznych konferencji, podgrup typu *PyLadies*, kodeksów postępowania, uznania dla różnorodności, podkastów i biuletynów — Python ma również własne pojęcie *zen*.

Zen Pythona

Wiele księżyców temu Tim Peters (jeden z wczesnych przywódców społeczności Pythona) usiadł i zaczął się zastanawiać nad tym, *co sprawia, że Python jest Pythonem*.

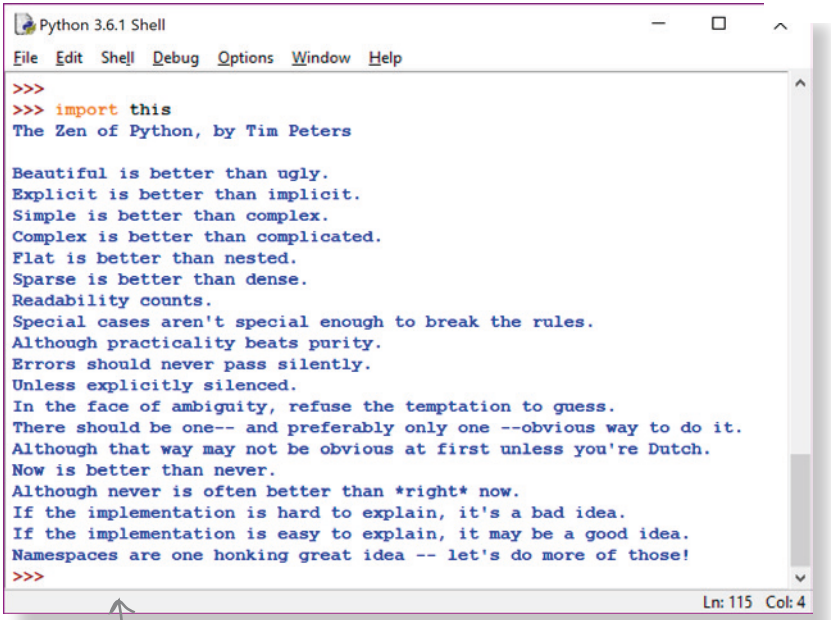
Odpowiedź przyszła do niego pod postacią *Zen Pythona* (ang. *Zen of Python*), którego treść możesz przeczytać w oryginale po uruchomieniu dowolnej wersji interpretera tego języka i wpisaniu w wierszu poleceń instrukcji:

```
import this
```

Zrobiliśmy to za Ciebie i na zrzucie ekranowym okna IDLE przedstawionym w dolnej części tej strony pokazaliśmy tekst wyświetlony w wyniku wykonania powyższego wiersza kodu[3]. Pamiętaj, aby czytać treść *Zen Pythona* co najmniej raz na miesiąc.

Wielu próbowało streścić *Zen Pythona*, aby uzyskać coś nieco łatwiejszego do przyswojenia. Nikt inny niż `xkcd` nie zbliżył się jednak nawet do tego celu. Jeśli masz połączenie z internetem, wprowadź przedstawiony poniżej wiersz kodu w swoim wierszu poleceń powłoki Pythona, aby zobaczyć (dosłownie), jak `xkcd` sobie z tym poradził:

```
import antigravity
```



Pamiętaj: czytaj to CO NAJMNIEJ raz na miesiąc.

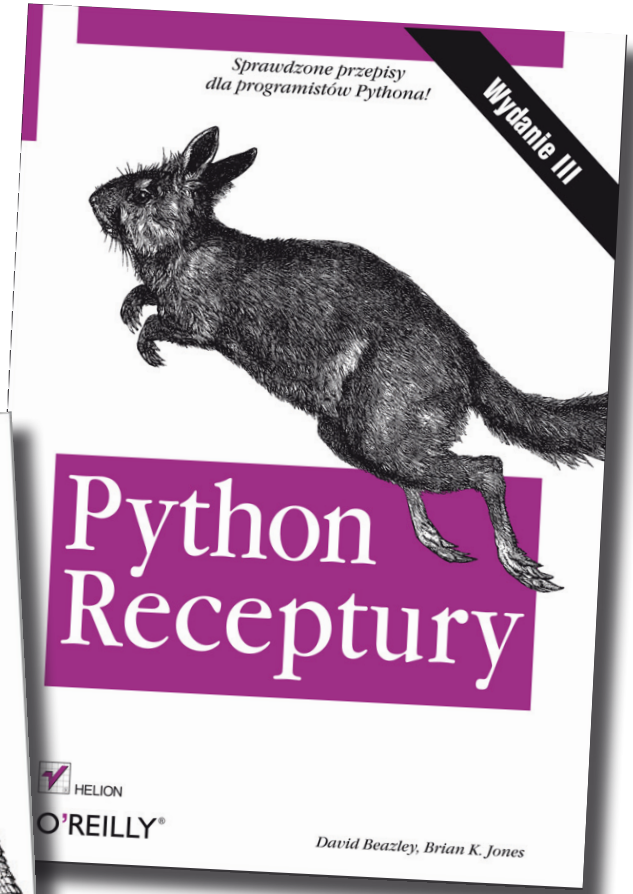
[3] Tłumaczenie treści *Zen Pythona* na język polski możesz znaleźć na przykład na stronie *http://misztal.edu.pl/logs/the-zen-of-python/* — *przy. tłum.*

Którą książkę powinienem przeczytać jako następną?

Nasze ulubione książki na temat Pythona

Wraz ze wzrostem popularności Pythona lawinowo rośnie liczba poświęconych mu książek. Ze wszystkich dostępnych na rynku pozycji wybraliśmy dwie publikacje wręcz niezbędne.

We wcześniejszym dodatku wspomnieliśmy już pracę Davida Beazleya. Pisząc tę książkę, David połączył swoje siły z Brianem K. Jonesem, aby udokumentować wspaniałą kolekcję receptur programowania. Jeśli kiedyś będziesz się zastanawiał nad tym, jak zrobić coś w Pythonie, przestań to robić i po prostu poszukaj odpowiedzi w książce „Python. Receptury".



Jeśli wgłębianie się w temat należy do Twoich ulubionych zajęć, przeczytaj tę doskonałą książkę. Znajdziesz w niej mnóstwo wiedzy, wszystko jest tu naprawdę dobre (i uczyni Cię lepszym oraz bardziej doświadczonym programistą Pythona).

Skorowidz

A

adnotacje funkcji, 162
adres
  IP, 359
  pętli zwrotnej, 211
  sieciowy, 198
aktualizacja
  aplikacji WWW, 348, 349
  funkcji log_request, 350
  funkcji view_the_log, 352
  licznika częstotliwości, 105, 106
algorytm mieszający, 100
analiza danych, 555
aplikacje WWW, 195, 198
  w chmurze, 238
  wdrażanie, 240
argument, 147, 154
  exc _value, 452
  exc_traceback, 452
  exc_type, 452
argumenty
  funkcji open, 247
  ze słowami kluczowymi, 171, 392
ataki sieciowe, 419, 422
atrybut, 14, 49, 311, 332
  secret_key, 409
  żądania sieciowego, 261
automatyczne przeładowania, 227

B

backend, 420
baza danych, 281–307
białe znaki, 40
biblioteka
  bokeh, 555
  Kivy, 560
  matplotlib/seaborn, 555
  numpy, 555
  pandas, 555
  scikit-learn, 555
  scipy, 555
  standardowa, 9, 10, 545
  tkinter, 547
BIF, built-in functions, 161
blok body, 214
błąd
  404, 234
  AttributeError, 483
  ConnectionError, 446
  FileNotFoundError, 424, 431
  FloatingPointError, 431
  InterfaceError, 418, 421
  KeyError, 115, 120, 371, 379
  ModuleNotFoundError, 176
  NameError, 321
  OverflowError, 431
  PermissionError, 431
  ProgrammingError, 448
  RuntimeError, 421, 423
  SQLError, 451, 453
  TypeError, 326
  ZeroDivisionError, 431
błędy
  bazy danych, 440
  czasu wykonania, 115, 414, 423, 425
  importowania, 176
  niedozwolonej metody, 225

C

chmura, 238, 532
ciasteczka, 367, 373
ciąg, 24
  URL, 207, 211
cudzysłów
  podwójny, 152
  pojedynczy, 152
częstotliwość, 104
część wspólna zbiorów, 128

D

dane, 243
  CSV, 479
  sieciowe, 557
  strukturalne, 95
  wyjściowe
    postać tabelaryczna, 265
    użycie kodu HTML, 274
    użycie silnika Jinja2, 276
  żądania, 256
datetime, 29
DB-API, 281, 284, 290
  kursor, 291
  połączenie z serwerem, 291
  sterownik bazy danych, 291
  właściwości połączenia, 290
  zamykanie połączenia, 295
  zapytania SQL, 292
debugger pdb, 549
debugowanie, 224, 549
Dedent Region, 27
definiowanie
  komprehencji słownika, 494
  reprezentacji klasy, 329
dekorator
  @app.route, 222, 385
  check_logged_in, 402, 406
  copy_current_request_context, 471
dekoratory funkcji, 207, 363, 385, 395, 407
dodawanie
  argumentów, 154
  metod, 318
  modułów, 524
dokument PEP, 153
dokumentacja, 12, 66
  funkcji, 162
domyślne wartości argumentów, 170
dostęp
  do danych formularza, 226
  do dokumentacji, 66
  do stron aplikacji, 408
  do URL-i, 382, 396
  do wrażliwych treści, 364
  do wybranych stron WWW, 383
  do złożonych struktur danych, 141
dunder
  enter, 338
  exit, 338, 345
  init, 324, 332, 338
  repr, 331
dwukropek, 16, 24, 98
działanie frameworka Flask, 203
dziedziczenie, inheritance, 311

E

Eclipse, 553
edytor, 57
escaping, 257

F

filtr, 495
Flask, 471
  dostęp do danych formularza, 226
  działanie, 203
  instalowanie, 202
  mechanizm sesji, 367
  renderowanie stron, 216
  renderowanie szablonów, 217
  tryb debugowania, 224
  tworzenie obiektu, 206
format
  danych, 484
  łańcucha znakowego, 486
formularz HTML, 213
  wyświetlanie, 218
framework
  aplikacji sieciowych, 201
  Flask, 202
funkcja, 145, 147
  append, 58, 59
  bool, 156
  check_logged_in, 384, 399, 403, 406
  check_status, 396
  date.isoformat, 11
  date.today, 11
  dir, 36
  do_search, 250, 467
  escape, 257, 270
  execute_slowly, 466
  getcwd, 10
  getenv, 10
  getuser, 369
  help, 36
  inner, 398
  len, 58
  list, 42
  log_request, 250, 253, 263, 298, 300, 304, 350, 440
  open, 246, 247
  outer, 398
  print, 15, 245, 263
  randint, 20, 23, 30, 32
  range, 25, 40
    koniec, 41
    krok, 41
    początek, 41
  render_template, 276
  repr, 329
  search4letters, 210
  set, 124, 129, 499
  sleep, 20, 23, 32, 420
  sorted, 109, 110, 129
  strftime, 11
  today, 14
  unescape, 11
  view_the_log, 273, 352, 356
  wrapper, 399
funkcje
  adnotacje, 162
  argumenty, 147, 154
  argumenty ze słowami kluczowymi, 392
  dekorowanie, 207
  dokumentacja, 147, 162
  dowolna liczba i typ argumentów, 394
  generatorowe, 512–515
  importowanie, 9
  instrukcja return, 156
  lista argumentów, 390
  obsługa argumentów, 401
  ogólnego przeznaczenia, 165
  przekazywane do funkcji, 386
  sygnatury, 409
  szablon, 148
  tworzenie, 149, 166–168
  typy argumentów, 148
  udostępnianie w sieci, 209
  wbudowane, 161
  wywoływanie, 9, 150
  zagnieżdżone, 388, 400, 409
  zwracane z funkcji, 389
  zwracanie wartości, 156–159

G

generator, 508, 511, 517
generowanie
  losowych liczb, 30
  wyników, 229
graficzne interfejsy użytkownika, 547

H

hermetyzacja, encapsulation, 311
HTML, 213
HTTP, HyperText Transfer Protocol, 365

I

IDE, integrated development environment, 3
identyfikator obiektu funkcji, 386
IDLE, 3, 7, 19, 151
implementacja
  referencyjna CPythona, 561
  protokołu, 340
importowanie, 29
  dekoratora, 404
  funkcji, 9, 36
indeksowanie listy, 89
Indent Region, 27
inicjalizowanie
  atrybutów, 323–325
  klasy, 341
instalowanie
  frameworka Flask, 202
  narzędzi programistycznych, 190
  pakietów, 182
  Pythona
    w systemie Linux, 527
    w systemie Mac OS X, 525
    w systemie Windows, 522
  serwera bazy danych, 283
  sterownika bazy danych, 285
instancja obiektu, 312
instrukcja
  elif, 17
  else, 16
  from time import sleep, 36
  if, 15, 239
  import
    checker, 397
    mysql.connector, 303, 360
    time, 36
  INSERT, 289, 463
  raise, 421
  return, 156
  SELECT, 354, 359, 464
  try, 424, 426, 468
  with, 247, 305, 310, 335
instrukcje
  importowania, 29, 36, 177, 303, 397
  sterujące, 16
  warunkowe, 17
interpreter, 7, 8, 19
iterowanie, 20, 477
  określoną liczbę razy, 25
  po kluczach i wartościach, 108
  po sekwencji obiektów, 24
  po słowniku, 107

J

język
  HTML, 213
  SQL, 289
Jinja2, 214, 276
Jupyter Notebook, 554

K

klasa, 309, 346
  ConnectionError, 446
  CountFromBy, 313, 314
  Flask, 205
  Thread, 466
  UseDatabase, 340
klasy
  atrybuty, 311, 332
  dodawanie metod, 318
  menedżer kontekstu, 340
  metody, 311, 332
  obiekty, 332
  stan, 311
  wartości domyślne, 330
  zachowanie, 311
klauzula except, 424, 426, 441
klawisz F5, 6
klucz, 98
  logged_in, 379
  user, 370, 372
kod
  demontujący, 346
  HTML, 213
  oczekujący, 463
  równoległy, 465
  ściśle sprzężony, 442
  współdzielony, 336, 383
  źródłowy strony, 255
kody stanu HTTP, 222
  informacyjne, 222
  powodzenia, 222
  przekierowania, 222
kolekcja obiektów, 50
  niezmienna, 51
  uporządkowana, 50, 51, 54
komentarz, 151
kompilacja, 7
komprehencja, 477, 491–493, 502
  listy, 493, 504, 517
  słownika, 493, 517
  zbioru, 504, 517
komunikat błędu, 192, *Patrz także* błąd
    klienta, 222
    serwera, 222
    Traceback, 6, 115, 470
konfigurowanie
  bazy danych, 341
  aplikacji, 536
konsola MySQL, 287
konstrukcja
  dunder name dunder main, 240
  request.form, 227
  try-except, 424, 426, 468
konwersja, 486, 496
kopiowanie struktury danych, 73
krotka, tuple, 51, 132
  pojedynczy łańcuch znakowy, 134
  tworzenie, 132
kursor, 291

L

liczba, 48
  losowa, 23, 30
licznik częstotliwości, 105, 106
lista, list, 13, 50
  indeksowanie, 89
  kopiowanie, 72, 73
  metoda
    append, 65, 71
    copy, 73
    extend, 64, 71
    insert, 65, 71
    pop, 63, 71
    remove, 62, 71
  niemodyfikowalna, *Patrz* krotka
  odczytywanie danych CSV, 479
  pętla for, 86
  plist, 69
  stosowanie komprehencji, 493
  usuwanie obiektów, 62
  używanie, 91
  wartość
    koniec, 76
    krok, 76, 79
    początek, 76
  wstawianie obiektu, 63, 65
  wycinek, 77, 79, 81, 85, 88
listy
  argumentów, 390, 391
  krotek, 292
  literalne, 55
  łańcuchów znakowych, 269
localhost, 211
logowanie, 374–376, 380
  atrybutów żądania, 261
  danych, 253, 264
  do bazy danych, 450
  wiersza rozdzielonych danych, 262
lokalizacje
  biblioteki standardowej, 174
  pakietów interpretera, 174, 178
losowy odcinek czasu, 30

Ł

łańcuch
  znakowy, 24, 48, 268, 483
  metod, 517
łączenie
  struktur danych, 135
  wywołań metod, 483
  zbiorów, 126

M

manipulowanie danymi, 243
mapa, 96
MariaDB, 282
mechanizm
  importowania, 29
  sesji, 367
  wątków, 468
menedżer
  kontekstu, 306, 332, 337–340, 346
  pakietów, 525
metoda, 49, 147, 311, 332
  __enter__, 343, 344
  __eq__, 324
  __exit__, 345
  __init__, 324, 327, 332, 342
  __repr__, 332
  app.run, 211, 240
  append, 65
  close, 245
  conn.commit, 295
  copy, 73
  cursor.execute, 292, 294
  cursor.fetchall, 292
  difference, 127
  extend, 64
  GET, 222
  insert, 65
  intersection, 128
  items, 110, 121
  pop, 63
  POST, 222, 223
  readlines, 267, 269
  remove, 62
  setdefault, 120
  split, 268, 270
  union, 126
moduł, 145, 173, 183
  collections, 545
  contextlib, 337
  csv, 479
  datetime, 8, 11
  DBcm, 443, 448
  doctest, 548
  Flask, 202
  functools, 545
  itertools, 545
  os, 10
  pprint, 139
  random, 30, 32
  sys, 10, 430
  time, 32

unittest, 548
vsearch, 178
modyfikacja funkcji
log_request, 351
view_the_log, 354
MySQL, 282

N

narzędzia programistyczne, 559
narzędzie
pip, 182, 552
pytest, 189, 191
setuptools, 178
nawiasy
klamrowe, 16, 123
kwadratowe, 13, 66, 74, 99
okrągłe, 134, 508
trójkątne, 256
nazwa
atrybutu, 322
self, 317
this, 317
notacja
CamelCase, 312
kropkowa, 14, 316
nawiasów kwadratowych, 75
wycinka, 77

O

obiekt, 48, 312, 332, 386, 542
funkcji, 386, 409
request, 227
żądania, 256, 259
obiekty
atrybuty, 313
metody współdzielone, 313
obsługa
argumentów, 401
bazy danych, 282, 301, 336
logowania, 376
URL-a, 382
wyjątków, 413, 415, 457
błędy bazy danych, 440
konstrukcja try-except, 424
moduł sys, 430
przechwytywanie wszystkich wyjątków, 428
SQLError, 451, 453
odczytywanie
danych CSV, 479, 480
z bazy danych, 357
odpowiedź sieciowa, 198
odpytywanie bazy danych, 463
odwołania do atrybutów, 332
okno
Python Shell, 3, 22
Untitled, 3
OOP, object-oriented programming, 49, 311
operacje
na datach, 11
wejścia-wyjścia, 420
operator
in, 15, 56, 117, 119
mnożenia, 87
not, 60
przypisania, 13
operatory not in, 119
opóźnienie, 422
osadzanie logiki wyświetlania, 275
oznaczanie bloków kodu, 16

P

pakiet, 178
pamięć podręczna, cache, 295
para klucz-wartość, 52, 96
parametr, 147
PEP, Python Enhancement Proposal, 153
PEP 8, 166, 188, 263
pętla
for, 20, 24, 36, 86, 492, 517
while, 24, 36
pliki
CSV, 478
dystrybucji, 178–181
konfiguracji, 179
logu, 253, 254
odczytywanie danych, 246
otwieranie, 245
przetwarzanie, 245
zamykanie, 245
pobranie stanu, 369
podprogram, 147
polecenie, *Patrz także* instrukcja
cd, 175
describe log, 289
File/New File, 57
py -3, 175
quit, 289
polimorfizm, 311
połączenie
z bazą danych, 418, 422
z serwerem, 291
pomoc, 31
dotycząca funkcji, 41
na temat metody, 66
ponowne użycie kodu, 145
port, 211
PostgreSQL, 282
powłoka, 22, 36
poziom wcięcia, 18
procedura, 147
program dunder.py, 239
programowanie zorientowane obiektowo, OOP, 49, 311
protokół
HTTP, 365
zarządzania kontekstem, 248, 310, 335, 337
przechowywanie danych, 243, 265, 300
przechwytywanie
błędów, 424
wszystkich wyjątków, 428
przeglądanie
logu, 254
pełnego logu, 258
przeglądarka internetowa, 198
przekazywanie funkcji, 386
przekierowanie, 235
przekształcanie danych
słownik list, 485
surowych danych, 274, 275
wybrany format, 484
przenoszenie danych do bazy, 355
przestrzeń nazw, 29
przetwarzanie
danych, 270
listy argumentów, 391
słownika argumentów, 393
URL-i, 509
generator, 510
komprehencja listy, 509
przyjmowanie
listy argumentów, 390
słowników argumentów, 392
przypisanie
pozycyjne, 171
struktury danych, 14
za pomocą słowa kluczowego, 171
zmiennej, 13
PyCharm, 553
PyPI, Python Package Index, 202
Python 2, 540
Python Shell, 22
PythonAnywhere, 238, 531

R

refaktoryzacja, 119
referencja, 184
relacyjna baza danych, 281
renderowanie szablonów, 217
REPL, 4
repozytorium, 12

reprezentacja
klasy CountFromBy, 328
obiektu, 329
rodzaje struktur, 47
rozpoznawanie problemów, 416
rozszerzenie py, 5
równoległe wykonywanie kodu, 546

S

sekwencja, 24
ucieczki, 257
self, 317–320, 344
serwer, 198, 201
bazy danych MySQL, 282, 283
WWW, 211, 366
serwis PythonAnywhere, 531
sesja, 367, 368
frameworka Flask, 368
zarządzanie logowaniem, 374
silnik
Jinja2, 214, 276
szablonów, template engine, 213
składnia
adnotacji, 162
z notacją kropkową, 58
słownik, dictionary, 52, 91
inicjalizowanie kluczy, 115
iterowanie, 107
iterowanie po kluczach i wartościach, 108
kolejność wprowadzania, 99
licznik częstotliwości, 105, 106
metoda
items, 110, 121
setdefault, 120
odczytywanie danych CSV, 480
pary klucz-wartość, 96
pętla for, 108
pobranie stanu, 369
sprawdzanie przynależności, 117
wyszukiwanie wartości, 100
słowniki
argumentów, 392, 393
przechowywania częstotliwości, 104
słowników, 137, 267
słowo kluczowe
async, 546
await, 546
class, 332
def, 147, 149
raise, 444
return, 147, 156
self, 317–320, 344
this, 317
sortowanie, 544
specyfikacja aplikacji WWW, 200
specyfikatory formatów, 543
społeczność Pythona, 563
sprawdzanie
stanu, 377
surowych danych, 256
zgodności kodu, 189, 193
SQL, 282
instrukcja INSERT, 289, 463
instrukcja SELECT, 354, 359, 464
SQLi, SQL injection, 419
stan, state, 311, 368, 380
aplikacji, 366
starter aplikacji, 534, 535
sterownik bazy danych, 282, 285
MySQL-Connector/Python, 286
stosowanie DB-API, 281
struktura danych, 13, 19
krotka, 51, 132
lista, 50
słownik, 52
zasada 80/20, 53
zbiór, 53, 123
strumień pliku, file stream, 245, 246
submoduł datetime, 8
surowe dane, 256
symbol, *Patrz* znak
szablon, 275
bazowy, 214
funkcji, 148
sztuczne opóźnienie, 420

Ś

ścieżka wyszukiwania, 173
środowisko programistyczne
Eclipse, 553
IDLE, 3
REPL, 4

T

tabela
log, 300
modyfikowalna, 136
tablica, 50
mieszająca, 96
wyszukiwania, 97
technologie bazodanowe, 558
testowanie, 190, 548
tryb
debugowania, 224
dołączania, append mode, 245
tworzenie
aplikacji WWW, 197, 556
bazy danych, 287
dekoratora funkcji, 395, 397, 407
formularza HTML, 213
funkcji, 149, 166–168
funkcji do_search, 210
klas, 309
krotek, 132
list, 54, 55
menedżera kontekstu, 306, 337–340
modułu, 173
obiektów klas, 206, 312
pliku dystrybucji, 179, 180
pliku konfiguracji, 179
tabel, 287
własnych wyjątków, 444, 449
zbiorów, 124
zbioru z sekwencji, 124
typ zmiennej, 13, 19

U

uchwyt pliku, file handle, 245
udostępnianie funkcji, 209
układ pozycyjny, 171
unikatowe ciasteczko, 373
uporządkowana kolekcja obiektów, 54
URL, 207, 211, 382
uruchamianie
aplikacji WWW, 220
kodu, 8
UseDatabase, 306, 340
usługa PythonAnywhere, 240
usuwanie obiektów z listy, 62
uwierzytelnieni użytkownicy, 364
używanie
bazy danych, 281, 293, 296
danych żądania, 227
dekoratorów, 411
generatora, 510, 511
kodu HTML, 274
listy, 56, 91
komprehencji listy, 509
menedżera pakietów, 525
sekwencji ucieczki, 257
słownika słowników, 267
wątków, 466, 474
wielokrotne kodu, 145
zbioru, 131
użytkownik autoryzowany, 366

W

warstwa abstrakcji, 442
wartości
argumentów, 170
boolowskie, 116
domyślne klasy, 330
portów, 211

wartość, 98
koniec, 41, 76
krok, 41, 76, 79
logiczna
False, 116, 157
True, 116, 157
początek, 41, 76
zmiennej __name__, 237
wątki, 466
wcięcie, 16, 24, 40
wdrażanie aplikacji WWW, 240, 529
wiersz poleceń, 21, 175, 552
WingWare, 553
wirtualne środowiska programistyczne, 541
wizualizacja złożonych struktur danych, 140
właściwości połączenia, 290
wprowadzanie danych do aplikacji, 253
współdzielenie
kodu, 336, 383
zachowania, 332
wstrzyknięcie
kodu SQL, SQLi, 416, 419
skryptu, XSS, 416, 419
wstrzymanie wykonania, 28
wtyczka pep8, 192
wycinek, slice, 79, 81, 85
wycinki pętli for, 88
wyjątek, 413, *Patrz także* błąd, obsługa wyjątków
wyjątki własne, 444, 449
wylogowywanie, 377, 380
wyodrębnianie danych, 500, 501
wyświetlanie
formularza HTML, 218
złożonych struktur danych, 139
wywoływanie
funkcji, 9, 150, 421
logującej, 250
modułów wbudowanych, 14
przekazanej, 387
metody, 14, 316, 317
przez referencję, 186
przez wartość, 185
wzorzec programowania, 489

X

XSS, Cross-site scripting, 419

Z

zachowanie, behavior, 311
zakresy liczb całkowitych, 42
zamykanie połączenia, 295
zapis pliku, 5
zarządzanie
kontekstem, 248, 305, 335, 338
logowaniem, 374
zasada 80/20, 53
zasięg zmiennej, 321
zastępowanie
operatora in, 119
instrukcji with, 310
zbiór, set, 53, 123
część wspólna, 128
komprehencja, 505
łączenie, 126
metoda
difference, 127
intersection, 128
union, 126
pusty, 160
tworzenie, 124
zestawy, 16, 19
zagnieżdżone, 18
zgłaszanie wyjątku, 446, 453
zliczanie wystąpień, 111
złożone struktury danych, 139
uzyskiwanie dostępu, 141
wizualizacja, 140
wyświetlanie, 139
zmienna, 13
__name__, 237
iteracyjna pętli, 24
logged_in, 366
zmienne
globalne, 366
Jinja2, 230
zasięg, 321
znacznik
<form>, 222
<table>, 274
<td>, 274
<th>, 274
<tr>, 274
czasu, timestamp, 288
znak
*, 390, 391
@, 207, 237, 471
<, 257
>, 257
cudzysłowu, 152
dwukropka, 16, 24, 98
nowego wiersza, 262, 482
pionowej kreski, 262
spacji, 40
tabulacji, 40
ukośnika wstecznego, 77
zachęty, 10, 22
zakończenia wiersza, 262
znaki **, 392

Ź

źródło strony, 255

Ż

żądanie sieciowe, 198, 365

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA